白话强化学习与PyTorch

高扬 叶振斌 著　　万娟 插画

REINFORCEMENT LEARNING
PYTORCH

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

VISIT…

内 容 简 介

本书以“平民”的起点，从“零”开始，基于 PyTorch 框架，介绍深度学习和强化学习的技术与技巧，逐层铺垫，营造良好的带入感和亲近感，把学习曲线拉平，使得没有学过微积分等高级理论的程序员一样能够读得懂、学得会。同时，本书配合漫画插图来调节阅读气氛，并对每个原理都进行了对比讲解和实例说明。

本书适合对深度学习和强化学习感兴趣的技术人员、希望对深度学习和强化学习进行入门了解的技术人员及深度学习和强化学习领域的初级从业人员阅读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

白话强化学习与 PyTorch / 高扬，叶振斌著. —北京：电子工业出版社，2019.9

ISBN 978-7-121-36747-2

Ⅰ. ①白… Ⅱ. ①高… ②叶… Ⅲ. ①人工智能—程序设计 Ⅳ. ①TP18

中国版本图书馆 CIP 数据核字（2019）第 111731 号

责任编辑：潘 昕

印 刷：三河市双峰印刷装订有限公司

装 订：三河市双峰印刷装订有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787 × 980 1/16 印张：24 字数：590 千字 彩插：2

版 次：2019 年 9 月第 1 版

印 次：2019 年 9 月第 1 次印刷

定 价：99.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819，faq@phei.com.cn。

前　言

当人工智能的热潮席卷全世界的时候，当我们一次又一次为各种人工智能新作而惊叹的时候，我们不约而同地将目光投向了深度学习领域。我们对这个既新鲜又传统的领域有一种憧憬，甚至我们之中的许多人会在某一刻认为这可能是未来机器觉醒前的萌芽。

说深度学习新鲜，是因为它在近几年的发展确实获得了令人瞩目的成果，不论是在计算机视觉（Computer Vision，CV）领域，还是在自动语音识别（Automatic Speech Recognition，ASR）领域，抑或推荐系统应用领域，都迈上了一个前所未有的新台阶。这当然得益于计算机硬件处理能力的提升，包括 CPU 的速度、内存的大小、磁盘的容量；得益于新型并行计算框架的逐步成熟，例如 CUDA（Compute Unified Device Architecture）；得益于广大热衷于贡献高质量开源软件系统的大厂的扛鼎力作，例如 TensorFlow、PyTorch、MXNet 等优秀框架的不断涌现。这些无一例外，都成为推动一个行业在全国乃至全世界发展的重要因素。

说深度学习传统，是因为它最为基本的理论部分，是在数十年前就逐步有人提出并进行完善的一套理论体系。一位朋友曾告诉我，在他研究生毕业的时候（应该是在 21 世纪初），有不少同学的研究方向是人工智能和神经网络，而其中的大多数人，不是进了国有研究所，开始了日复一日“读 Paper、写 Paper”的过程，就是进了保险公司、证券交易所等企业——做什么呢？也不是做自己的老本行，而是做报表、做数据分析、写报告。

当然，这些同学算是比较“幸运”的，还有相当一部分干脆改了行，去往一些企业的开发岗位，做 Java 开发或者数据库架构之类的工作。我完全没有轻视这些岗位和企业的意思，我只是单纯地觉得，在大学中花了那么多年走出来的一条路，没办法再走下去，是一件非常可惜的事情。而在导致这一现象的众多客观因素中，前面提到的计算能力、框架成熟度、软件包可用性等起着决定性的作用。

不过，我们赶上好时候了。除落地应用外的大部分基础问题，都已经由那些世界一流的数据科学家帮我们解决，并封装成了一个个开源或非开源的软/硬件产品。

其实，除了深度神经网络在 CV[①]、ASR[②]、NLP[③]等领域的应用，近几年，深度学习在另外一个领域有着炫目的成绩，那就是深度强化学习[④]领域。别的不说，大名鼎鼎的 AlphaGo 和 AlphaGo Zero 就是典型的深度强化学习应用。从感性的角度看，深度强化学习似乎是由“深度学习”和“强

① CV（Computational Vision），计算机视觉。
② ASR（Automatic Speech Recognition），自动语音识别。
③ NLP（Natural Language Processing），自然语言处理。
④ 深度强化学习，Deep Reinforcement Learning（DRL）。

化学习”两个领域结合而成的——这个说法应该不算错。

所谓深度学习，一般是指基于深度神经网络的一系列应用，而这个领域应用的丰富程度也是非常高的。因为深度神经网络有着超高的 VC 维 [①]，所以理论上它几乎可以用来实现任何从 x 到 $f(x)$ 的映射关系。

强化学习所研究的事情就更有意思了。强化学习研究的主要是如何让机器人在一个预先定义好的环境中，通过自己学习来逐步形成或学会一套成熟的、高质量的自动化行为策略，这也是一直以来世界一流高校的人工智能专业设立并努力研究的一个神秘而又让人充满幻想的领域。

为了让广大读者能够近距离接触强化学习和深度学习结合应用的案例，让这样一个有趣且有着光明前景的领域以更加平易近人的姿态出现，在这本书中，我将用具有高中数学水平的读者朋友能够理解的语言，和大家好好分享一下如何用深度学习的利器 PyTorch 来完成人工智能机器人自我进化的落地过程。

本书特色

可以说，我们处于人工智能时代的起点。这是一个非常好的时代，那么多优秀的开源软件框架，为每一个程序员提供了门槛足够低的学习工具。

本书以“平民”的起点，从“零”开始介绍深度学习的技术与技巧，让读者有良好的带入感和亲近感，并通过逐层铺垫，把学习曲线拉平，使得没有学过微积分等高级理论的程序员朋友一样能够读得懂、学得会。同时，本书配合漫画插图来调节阅读气氛，并在每个原理讲解的部分都提供了对比和实例说明，相信每位读者都能有轻松、愉悦的阅读体验。

如果你的英文阅读能力比较强且有比较高的学术素养，如果你能够以英文原版文献（不论是公开课、论文，还是经典教材）作为首选的阅读资料，那么，仍然推荐你阅读英文原版内容，因为那将让你更为直接地接触相关知识。

基于本书的定位，在描述讨论对象的时候，我通常不会使用学术级别的词汇，而是尽可能使用程序员能够读懂的语言。因此，与经典的阅读材料相比，本书叙述的内容难免存在一些偏颇，甚至可能为了言简意赅地讲明原理而有失严谨。另外，在一个理论出现或者相关论文发表之后，从我把它读懂、写清楚到图书出版，至少需要一年，甚至可能更久，而这对于一个在领域前沿工作的技术人员来说，时间显得有些长。因此，对于学习能力比较强的读者来说，本书作为工程实现参考书更为合适。

① VC 维（Vapnik-Chervonenkis Dimension），描述网络对空间的划分能力。VC 维越高，划分能力就越强。

读者对象

本书适合如下人员阅读。

- 对深度学习和强化学习感兴趣但完全不了解的技术人员。
- 对深度学习和强化学习比较感兴趣的技术人员。
- 深度学习和强化学习领域的初级从业人员。
- 希望对深度学习和强化学习进行入门了解的技术人员。

如何阅读本书

本书是我写作的“白话”系列图书的第三本，与《白话大数据与深度学习》《白话深度学习与TensorFlow》互为姊妹篇。但是，本书不依赖另外两本书的内容，是独立成册的。

本书的主要内容如下。

- 第 1 章到第 5 章，讲的是传统强化学习的研究目标与脉络，主要介绍了从一个程序员的角度怎么理解强化学习最为轻松，侧重于理解方式的诱导。
- 第 6 章到第 11 章，是本书的核心内容，讲的是深度学习的原理、PyTorch 框架的基本知识及深度强化学习的常用算法模型。
- 第 12 章到第 15 章，讲的是扩展性的知识。例如，其他有助于训练模型的算法思路，第三方工具插件，可供实验的环境，一些有趣的强化学习算法和观点，甚至模型落地过程中的优化与压缩。
- 附录 A 中详细记载了本书涉及的各种软件环境的安装和配置过程。

套用一句时髦的广告语：“深度强化学习从入门到落地，读这一本就够了。”

勘误与支持

由于作者的水平有限，编写时间仓促，书中难免会出现错误或者不准确的地方，恳请读者批评指正。

如果你有更多的宝贵意见，欢迎扫描下方的二维码，关注“智能工厂”微信公众号和我们进行互动讨论。你也可以在这里找到本书配套代码的下载地址和 QQ 讨论群的相关信息。关注大数据尖端技术发展，关注“智能工厂”。

本书代码的下载地址为 https://github.com/GAOYANGAU/DRLPytorch。
同时，你也可以发送电子邮件至 77232517@qq.com 与我联系。

期待得到你的真挚反馈，让我们在技术道路上互勉共进。

高　扬
2019 年 3 月
于 珠海

目　录

传统篇

第 1 章　强化学习是什么 2
1.1　题设 3
1.1.1　多智能才叫智能 5
1.1.2　人工智能的定义 5
1.2　强化学习的研究对象 7
1.2.1　强化学习的应用场合 7
1.2.2　强化学习的建模 11
1.3　本章小结 17
第 2 章　强化学习的脉络 18
2.1　什么是策略 18
2.2　什么样的策略是好的策略 19
2.3　什么是模型 21
2.4　如何得到一个好的策略 23
2.4.1　直接法 23
2.4.2　间接法 25
2.5　马尔可夫决策过程 29
2.5.1　状态转移 30
2.5.2　策略与评价 31
2.5.3　策略优化 36
2.6　Model-Based 和 Model-Free 36
2.6.1　Model-Based 36
2.6.2　规划问题 37
2.6.3　Model-Free 38
2.7　本章小结 38
第 3 章　动态规划 40
3.1　状态估值 40
3.2　策略优化 42
3.3　本章小结 43
第 4 章　蒙特卡罗法 45
4.1　历史由来 45
4.2　状态估值 47
4.3　两种估值方法 49
4.3.1　首次访问蒙特卡罗策略估值 49
4.3.2　每次访问蒙特卡罗策略估值 49
4.3.3　增量平均 50
4.4　弊端 51
4.5　本章小结 52
第 5 章　时间差分 53
5.1　SARSA 算法 53
5.1.1　SARSA 算法的伪代码 54
5.1.2　SARSA 算法的优点和缺点 57
5.2　Q-Learning 算法 57
5.2.1　Q-Learning 算法的伪代码 58
5.2.2　Q-Learning 算法的优点和缺点 59
5.3　On-Policy 和 Off-Policy 61
5.4　On-Line 学习和 Off-Line 学习 62
5.5　比较与讨论 63
5.6　本章小结 65

现代篇

第 6 章　深度学习 68
6.1　PyTorch 简介 69
6.1.1　历史渊源 70
6.1.2　支持 71
6.2　神经元 74
6.3　线性回归 77
6.4　激励函数 80
6.4.1　Sigmoid 函数 81
6.4.2　Tanh 函数 82
6.4.3　ReLU 函数 82
6.4.4　Linear 函数 83
6.5　神经网络 84
6.6　网络训练 85
6.6.1　输入 86
6.6.2　输出 86
6.6.3　网络结构 87
6.6.4　损失函数 88
6.6.5　求解极小值 90
6.6.6　线性回归 90
6.6.7　凸函数 93
6.6.8　二元（多元）凸函数 98
6.6.9　导数补充 101
6.6.10　导数怎么求 103
6.6.11　“串联”的神经元 105
6.6.12　模型的工作 107
6.6.13　理解损失函数 108
6.7　深度学习的优势 108
6.7.1　线性和非线性的叠加 109
6.7.2　不用再提取特征 111
6.7.3　处理线性不可分 112
6.8　手写数字识别公开数据集 114
6.9　全连接网络 117
6.9.1　输入与输出 118
6.9.2　代码解读 119
6.9.3　运行结果 125
6.10　卷积神经网络 125
6.10.1　代码解读 125
6.10.2　理解卷积神经网络的结构 132
6.10.3　卷积核的结构 134
6.11　循环神经网络 135
6.11.1　网络结构 136
6.11.2　应用案例 140
6.11.3　代码解读 143
6.12　其他注意事项 148
6.12.1　并行计算 148
6.12.2　梯度消失和梯度爆炸 152
6.12.3　归一化 157
6.12.4　超参数的设置 159
6.12.5　正则化 161
6.12.6　不唯一的模型 170
6.13　深度神经网络的发展趋势 171
6.14　本章小结 178
第 7 章　Gym——不要钱的试验场 180
7.1　简介 180
7.2　安装 182
7.3　类别 183
7.4　接口 188
7.5　本章小结 191
第 8 章　DQN 算法族 192
8.1　2013 版 DQN 192
8.1.1　模型结构 192
8.1.2　训练过程 195
8.1.3　Replay Memory 197
8.1.4　小结 198

8.2 2015 版 DQN 198
8.2.1 模型结构 198
8.2.2 训练过程 199
8.2.3 Target 网络 200
8.2.4 小结 201
8.3 Double DQN 201
8.3.1 模型结构 202
8.3.2 训练过程 202
8.3.3 效果 203
8.3.4 小结 204
8.4 Dueling DQN 204
8.4.1 模型结构 205
8.4.2 效果 207
8.4.3 小结 208
8.5 优先回放 DQN 208
8.6 本章小结 209
第 9 章 PG 算法族 211
9.1 策略梯度 211
9.2 DPG 213
9.3 Actor-Critic 214
9.4 DDPG 214
9.5 本章小结 218
第 10 章 A3C 219
10.1 模型结构 219
10.1.1 A3C Q-Learning 219
10.1.2 A3C Actor-Critic 222
10.2 本章小结 224
第 11 章 UNREAL 226
11.1 主任务 226
11.2 像素控制任务 227
11.3 奖励值预测 229
11.4 值函数回放 230
11.5 损失函数 231
11.6 本章小结 232

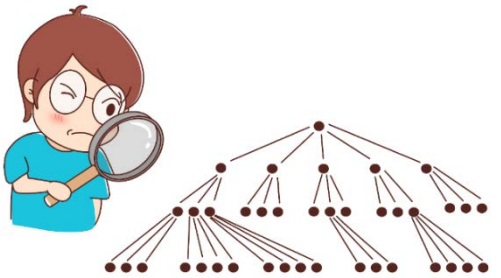
扩展篇

第 12 章 NEAT 236
12.1 遗传算法 237
12.1.1 进化过程 237
12.1.2 算法流程 238
12.1.3 背包问题 239
12.1.4 极大（小）值问题 247
12.2 NEAT 原理 255
12.2.1 基因组 255
12.2.2 变异和遗传 256
12.3 NEAT 示例 258
12.4 本章小结 262
第 13 章 SerpentAI 263
13.1 简介 263
13.2 安装和配置 264
13.3 示例 265
13.3.1 创建 Game Plugin 265
13.3.2 创建 Game Agent 268
13.3.3 训练 Context Classifier 271
13.3.4 训练 Agent 282
13.4 本章小结 286
第 14 章 案例详解 287
14.1 AlphaGo 287
14.1.1 AlphaGo 的前世今生 287
14.1.2 “深蓝”是谁 288
14.1.3 围棋到底有多复杂 290
14.1.4 论文要义 294

14.1.5 成绩 302
14.1.6 开源项目 303
14.2 AlphaGo Zero 304
14.2.1 改进之处 304
14.2.2 成绩 308
14.2.3 开源项目 309
14.3 试验场大观 311
14.3.1 StarCraftⅡ 311
14.3.2 VizDoom 320
14.3.3 Universe 323
14.3.4 DOTA2 324
14.4 本章小结 329
第 15 章 扩展讨论 331
15.1 TRPO 331
15.2 反向强化学习 332
15.3 模型压缩 333
15.3.1 剪枝 335
15.3.2 量化 336
15.3.3 结构压缩 337
15.4 本章小结 339

后记 341
附录 A 342
A.1 安装 Ubuntu 342
A.2 安装 CUDA 环境 347
A.3 安装 PyTorch 348
A.4 下载本书示例代码 349
A.5 安装 PyCharm 350
A.5.1 方法一 350
A.5.2 方法二 351
A.6 安装 Jupyter Notebook 351
A.7 安装相关 Python 依赖包 352
A.7.1 安装 Box2D 352
A.7.2 安装 MuJoCo 352
A.7.3 安装 SerpentAI 355
A.7.4 安装 Spritex 359
A.7.5 安装 StarCraftⅡ 360
A.7.6 安装 VizDoom 363
A.8 安装 OpenCV 364
A.9 Python 语言简介 364
A.9.1 安装 Python 365
A.9.2 Hello World 365
A.9.3 行与缩进 365
A.9.4 变量类型 366
A.9.5 循环语句 367
A.9.6 函数 368
A.9.7 模块 369
A.9.8 小结 369
A.10 本书涉及的主要开源软件版本 369
参考文献 371

传统篇



第 1 章　强化学习是什么

强化学习（Reinforcement Learning）是一个独立的机器学习研究领域。为了让大家有比较直观的感受，本书将对强化学习和机器学习的一些研究领域进行对比讨论。请原谅我的啰唆，因为有些时候，只有铺垫足够多，路走起来才足够稳。

先说说监督学习（Supervised Learning）。监督学习是一个比较传统的机器学习研究领域。简单概括，监督学习主要希望研究映射关系

$$y = f(x|\theta).$$

这其实是一个我们在中学时就见过的函数，唯一陌生的是 θ ——它是待定系数。具体来说，有一个确定了系数的函数和一个确定的 x，得到一个确定的 y 是顺理成章的事情。例如，有 $y = 2x + 3$，给我一个 x 的值，让我求出对应的 y 值，当然很容易——如果 $x = 2$，那么 $y = 7$，这里的 θ 就是 x 前面的系数 2 和后面的常数项 3。

如果我们实际观测到的值只有 $x = 2$、$y = 7$，但不知道 θ 的值，希望通过一系列科学的办法“反向”推导出 θ 的值，那么，这就属于机器学习的研究范畴了。在这个例子中，可以在给定足够数量的样本的情况下，也就是说，在有足够的 x 和对应的 y 的值的时候，尝试用线性回归的方法求出 θ 的值。将已知输入变量 x 和输出变量 y 的值作为参数，逐步求出待定系数 θ 的过程，就是训练（Training）过程。当通过这样一个过程学习到 θ 应该为什么值之后，如果再有 x 出现，就可以通过函数 $y = f(x|\theta)$ 计算出 y 的值。这就是监督学习的实际工作方式，这个例子就是典型的监督学习中的线性回归（Linear Regression）问题。这个过程的具体实现方法，将在 6.3 节中详细介绍。

再说说非监督学习（Unsupervised Learning）。非监督学习也是一个比较传统的机器学习领域。在机器学习入门阶段使用最多的算法 K-Means，就是一个典型的非监督学习算法。例如，在一个空间中，有很多的空间点向量（Vector）[①]，这些点在这个空间中的分布很可能是不均匀的。是否可以通过一个算法来求出它们各自的分布究竟聚集在哪些空间区域附近？是否可以找出这些空间区域的中心点的具体位置？答案是：可以。在这个过程中，我们只需要告诉计算机我们要把这些空间中的点分成几个聚类（Cluster），甚至不用告诉计算机我们要把这些点具体分成哪几个聚类，计算机就能对这些空间点向量进行聚类的划分。

如图 1-1 所示，空间中不均匀地分布着一些点，通过 K-Means 算法可以计算出它们分别聚集在三个点周围（或者说“聚成了三堆”）。

① 空间点向量可以是一维的，也可以是多维的。

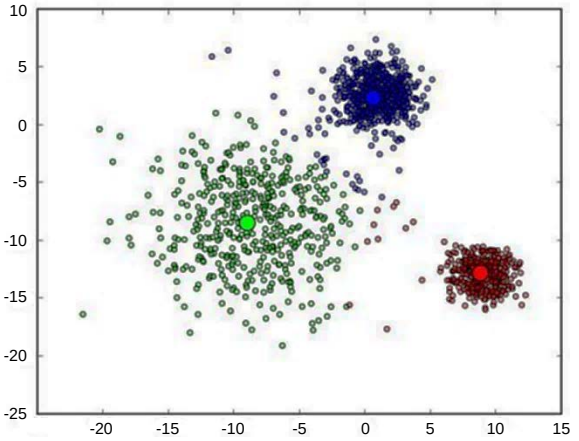


图 1-1　K-Means 算法示意

我们可以研究这三个聚类中的点都有哪些共性，也可以研究哪些点没有在这些聚类中或远离这些聚类中心。这样的应用场景在现实生活中是很多的。例如，能不能把用户的年龄、收入、贷款金额等数据绘制成这样的图像，通过聚类的方式找出用户人群的分布，然后针对不同的人群尝试设计相应的产品或业务，以分别满足他们的需要？这可比自己抱着脑袋绞尽脑汁冥思苦想要科学得多。

当然，我们还可以通过这种方法来研究那些不属于任何类的、飘在所有类之外的“离群点”。离群点的研究也是有价值的，因为它们太特殊了。它们为什么特殊？是数据收集系统本身的问题导致的特殊性，还是这个点本来就是一个“异类”（例如诈骗事件的特征点描述）？这些都是可以研究的思路。

显然，这个过程与研究 $y = f(x|\theta)$ 的过程（需要知道 x 和 y 的具体值，才能完成整个训练过程）不一样。在这个例子中，我们只需要知道 x 是什么就够了—— x 就是空间中的一个个点。

还有很多机器学习的研究领域，例如迁移学习（Transfer Learning）、生成对抗网络（Generative Adversarial Networks，GAN）等以不同建模方式或思考角度来解决问题的方法论。在这里，我之所以不愿意称它们为“算法”，而愿意称它们为“模型体系”或“方法论”，是因为它们确实已经各自拥有了一套较为完整的处理数据、建模、训练、调优的套路。它们中也包括本书的主角——强化学习。

1.1　题设

强化学习是为了解决哪些问题而设计的？在人工智能领域，我们其实一直希望机器人足够智能。这里的“智能”是指让计算机拥有比较好的推理、判断、分析能力，并基于这些能力自动进

化，最终胜任人类交给它们的任何任务——最理想的状态，就像阿诺德·施瓦辛格[①]和克里斯塔娜·洛肯[②]主演的《终结者》（*The Terminator*）系列电影里的未来机器人那样。可以说，和我同龄的“80 后”中的很多人都是在这类科幻作品的启蒙下，对人工智能产生了懵懂的感觉。《终结者》系列电影的第一部是 1984 年在美国上映的，看这部电影时我已经上大学了，大概在 2002 年。在那时的我看来，电影中的机器人简直无所不能——全天候工作、刀枪不入、变形自如……在那个时候，我脑子里对人工智能的概念是：像“终结者”这样的高智能人形机器人，应该是未来某个时刻的研究目标，也八成是大学或者实验室里正在研究而且没准儿哪天真的能研究出来的东西。说起来，那时我的想法还真是天真。

我再次看到与机器人相关的电影，已经是 2008 年了。那也是一部极为成功的作品——《机器人总动员》（*WALL-E*），在豆瓣上有 9.3 分的高分评价。当时，那部电影让我热泪盈眶。

随着年龄的增长，虽然我不会再像第一次看《终结者》时那么“激动”了，但仍然免不了偷偷地想：如果真的能造出这样的机器人，那就太好了。同时，我也开始有意识地寻找相关资料。我真的开始好奇：在清华、哈佛这样的大学里，人工智能专业的学生都在学什么？离实现这个目标还有多远？是不是有朝一日我们也能造出这样的机器人？

回过头来看，不得不承认，那时自己还是太年轻。其实，在有了多年的工作经历之后，我知道，虽然这两部科幻电影里面的机器人很难制造出来，但它们仍然属于人形机器人的研究范畴，涉及材料学、工程力学、大规模集成电路、高精度传感器工程学等众多前沿研究领域，而算法或者机器学习在其中只占很小的一部分——很小，很小。

在此，应该正式提出我的观点了：人形机器人不是强化学习研究的重点和难点，或者说，当前强化学习研究和落地的领域及其擅长解决的问题并不在人形机器人这个方面。不过，这并不会让我原本在强化学习领域积累的浓厚兴趣有一丝一毫的减弱，因为我仍然看到了基于海量样本训练的阿尔法围棋（AlphaGo）那令人惊叹的表现。

也许有些人会说，如今在互联网应用中已经出现了很多优秀的作品，不论是人脸识别（Face Recognition）、物体识别（Object Detective），还是智能相机的美图、美颜功能[③]等，不都是通过样本积累一点一点地进步，最终满足我们对产品的要求的迭代过程吗？刚刚说过的那些问题，能够通过强化学习来解决吗？两者有本质的区别吗？我想，在表达自己的观点之前，有些事情还是要和大家讨论清楚，这样我们在进入这本书的主要内容之前才能有足够的共识。

---

① 阿诺德·施瓦辛格（Arnold Schwarzenegger，1947 年 7 月—），政治家，美国加州前州长，健美运动员、力量举运动员，演员、导演、制片人。

② 克里斯塔娜·洛肯（Kristanna Loken，1979 年 10 月—），演员、制作人。

③ 这仍然是计算机视觉（Computer Vision）的研究范畴。

1.1.1　多智能才叫智能

到目前为止，不仅在学术界，甚至在工业界，仍然有不少人在争论——他们总会觉得某一种或某一类算法更为智能，或者更贴近智能的本质。

例如，据我所知，有很多人对本书的主角强化学习格外痴迷，觉得它才是实现人工智能的唯一靠谱途径（因为它的推理过程是那么优美和自然）。好吧，我不反对这些人对强化学习优美和自然的推理过程的称赞，因为这是事实。

也有相当一部分人认为，深度神经网络在形态上酷似人类的神经网络，是仿生学对人体神经结构的顶礼膜拜在计算机领域的完美落地。确实，这种结构在近些年的发展给我们带来了太多的惊喜与幻想。

还有一些人认为，只有像遗传算法这样的算法才能体现大自然优胜劣汰、适者生存的本质进化原理，因此有了“万般皆下品，唯有遗传高”的另类思路。我在工程中使用遗传算法的经验，确实让我多次体会到了遗传算法的优势，例如大大减少计算量、对于 NP 问题 [①]等复杂问题有着极高的提升效率的作用。

然而，刚刚提到的这些方法论或者算法族真的都体现了不容置疑且最为本质的智能体进化的原理特性吗？我并不这么认为。尤其是神经网络和遗传算法，它们仅仅是在“望文生义”的层面从人类神经网络和 DNA 遗传的角度进行的略显生硬的模仿，勉强做到了“形似”，谈不上理解并复现人类智慧进化的精华。要知道，在生命科学层面的认知上，人类还有很长的路要走。人类对自己的认知和了解的过程，其难度就像自举——自己抓着自己的头把自己举起来——岂止困难，简直不可能。现代医学对人类神经之间信号传递的了解仍是极为有限的，否则，能与人类神经嫁接的机械手臂早就问世了。人类对遗传因子理解的局限也是这样的。不说别的，仅转基因这样一个已经被科学家宣布“破译”的遗传因子应用子领域，还是令很多人忧心忡忡。这样一个看似被“破译”了的“上帝密码”在被我们诵读时带来的不和谐音符，已经让很多人看到了恶果——大家可以自己去网上找找有关孟山都 [②]的新闻。我个人从来不相信“阴谋论”，即使技术层面真的有问题，也是人类在自然面前的不自量力导致的。那么，我们还能拍着胸脯说自己理解了智慧生物的承载本质吗？从这个角度说这些技术是“形似”，都是客气的。

1.1.2　人工智能的定义

其实，在本书的开头就给人工智能下定义是不合适的，因为定义下得不好，很可能会误导读者。但是，对人工智能的定义没有基本的认识，也是一件很麻烦的事情，因为很可能没过多久就会有读者开始纠结于“这种人工智能够不够智能”“这个东西配不配叫人工智能”这样基本的而充满争议的问题了。

---

① Non-Deterministic Polynomial 问题，多项式复杂程度的非确定性问题。

② 孟山都是美国的一家跨国农业公司，其产品以农药、化肥、种子等农业科技产品为主。

需要注意的是，机器学习也好，人工智能也罢，这两个词汇本身就在无形中给人一种误导。它们让人们觉得，机器是自己在学习的，由人制造出来的这个“智能”的东西自己有分析、思考、进化的能力。因此，很多非业内人士会把这两个词背后的含义无限生发、放大、再放大。其中的“乐观”者觉得，在未来，机器人会非常厉害，什么都能做，什么都能自动完成，人类可以让机器人来伺候了；而“悲观”者则认为，人工智能会厉害到要么让人类“下岗”，要么直接统治地球、统治宇宙的地步……在我看来，这真的是一种误解——就像以理解畜力生产的方式去理解内燃机的工作原理，以理解老婆饼的方式来理解老婆……望文生义害死人。

前两年，我参与了一个图像识别项目，其中的一个项目组为了让一个神经网络能够认出视频中的物体，找了一群大学生来做打标签的工作（标出图片中的物体是什么东西、在什么位置），然后把数以万计打了标签的图片放到网络中去训练，从而让网络正确识别这些被标记物体的位置和分类。在这个过程中，超过 60% 的人工数（人×天）花在了标记样本上。当时，业界还有几个从事相关领域研究的大型机构，在这些机构的工作人员中间流传着这样一句话：有多少“人工”，就有多少“智能”。我当时说话可能更刻薄一些——这完全就是“人工”，哪儿来的“智能”。然而，不管怎么样，这种方法被验证了，它工作起来笨拙却有效，所有的公司和团队都可以用这种方式来复现深度神经网络应用中的监督学习过程。它让深度神经网络在迭代中不断提高自身的准确率（Accuracy），并最终在线上环境中自动工作，从而满足工程的需要。那么，这究竟算不算人工智能呢？

还是不着急回答这个问题。我们来追思计算机界的一位前辈——艾伦·麦席森·图灵 [①]。他曾经有一个很有趣的测试假设，被称为图灵测试（The Turing Test）。将测试者（一个人）与被测试者（一台机器）隔开，由测试者通过一些装置（例如键盘）向被测试者随意提问。进行多次测试后，如果有超过 30% 的测试者不能确定被测试者是人还是机器，那么这台机器就通过了测试，并被认为具有人类智能。

从广义上讲，如果人类通过有限的人机交互过程后发现，有相当比例的人无法判定与其交互的究竟是人还是机器，这台机器就通过了图灵测试，或者被称为具有人类智能。

乍听起来貌似没什么问题，不过不知道大家有没有注意到，在这里，图灵虽然提出了很多限制条件，但完全没有提到这台机器是否理解了交互过程的真正含义，也没有提到这台机器是否要跟人类一样具备这种在交互中确实存在的情绪。换句话说，在这个由机器和人进行的自动化互动过程中，只要人无法判断被测试者是否是机器，就可以判定这台机器拥有人类智能，至于什么理解含义啊、充满智慧啊，并不在考虑范围内。

这意味着什么呢？这意味着：这种人工智能仅仅是一个高质量的自动化过程，只要它能在人类关心的范畴高质量地完成作业就够了。没错，现在我们研究的所有人工智能项目，无论复杂还

① Alan Mathison Turing（1912 年 6 月 23 日—1954 年 6 月 7 日），英国数学家、逻辑学家，被称为计算机科学之父、人工智能之父。

是简单，无论高深还是浅显，只要它能够在应用范畴提高自动化程度、降低错误率，或者能在错误率与人类相当的情况下极大地提高处理效率，给出的结果和我们期望中由一个人来完成工作的结果很接近，就可以了。这些高质量的自动化过程，都应该属于人工智能的范畴。当然，这不能与我们所说的“智慧”画等号，因为此刻我们只是在就事论事地讨论“人工智能”。

1.2　强化学习的研究对象

我们来讨论一下强化学习的研究对象有哪些。

1.2.1　强化学习的应用场合

可能有心急的读者会问：你说了这么半天，该开始说说强化学习的研究对象都有哪些了吧？别急，这就开始。

强化学习的主要目的是研究并解决机器人智能体贯序决策问题。尽管我不喜欢直接把定义硬邦邦、冷冰冰地扔出来让大家被动接受，可还是免不了要在这里猛然给出“贯序决策”这么专业的词汇。好在，很快我就能通过例子把这个词汇给大家解释清楚了。既然大家要么是程序员，要么正走在程序员养成的路上，要么正看着其他人走在程序员养成的路上，那么，按照程序员的思维来理解强化学习将会更加顺畅。

把“贯序决策”翻译成“白话”就是：强化学习希望机器人或者智能体在一个环境中，随着“时间的流逝”，不断地自我学习，并最终在这个环境中学到一套最为合理的行为策略（如图 1-2 所示）。在这样一个完整的题设下，机器人应该尽可能在没有人干预的情况下，不断根据周围的环境变化学会并判断“在什么情况下怎么做才最好”，从而一步一步完成一个完整的任务。这样一系列针对不同情形的最合理的行为组合逻辑，才是一个完整的策略，而非一个简单而孤立的行为。

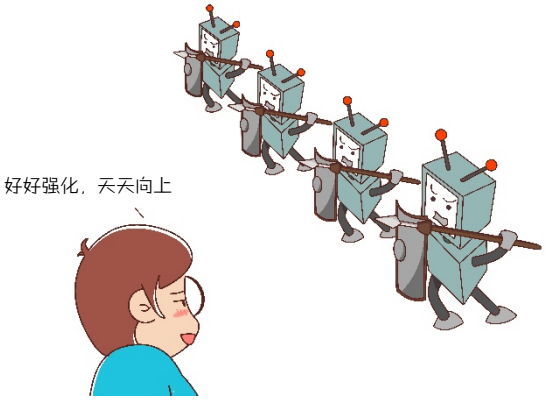


图 1-2　贯序决策

没错，这就是强化学习要研究并解决的问题。我想，可能已经有一些爱动脑筋的读者在心中暗暗反驳我了：这类问题一定要用强化学习来解决吗？我怎么觉得不用强化学习也能解决呢？

既然产生了这样一个争议，那我们先不去想是否一定要用人工智能这么“高大上”的工具来解决这类问题，而去想想自己以前有没有处理过类似的问题。如果一时想不起来，也没关系，请跟我看一个例子。

为了简化问题，假设我所在城市的道路是横平竖直的，各街区的边长相等，如图 1-3 所示。外侧的边框表示城墙，车辆无法通过。中间的方块表示街区，街区和街区之间就是道路。为了称呼方便，我们用英文字母 A ~ E 来标记纵向的道路，用数字 1 ~ 4 来标记横向的道路，并约定 A 大街和 1 大街交汇处的坐标为 A1（也就是 START 处）。圆圈中的数字表示我们用 GPS 或者其他手段测算出来的当前道路的拥堵程度，1 表示通畅，9 表示拥堵，也可以具象地将圆圈中的数字理解为通过某段道路需要 1 分钟到 9 分钟不等——总之，怎么理解简单怎么来。

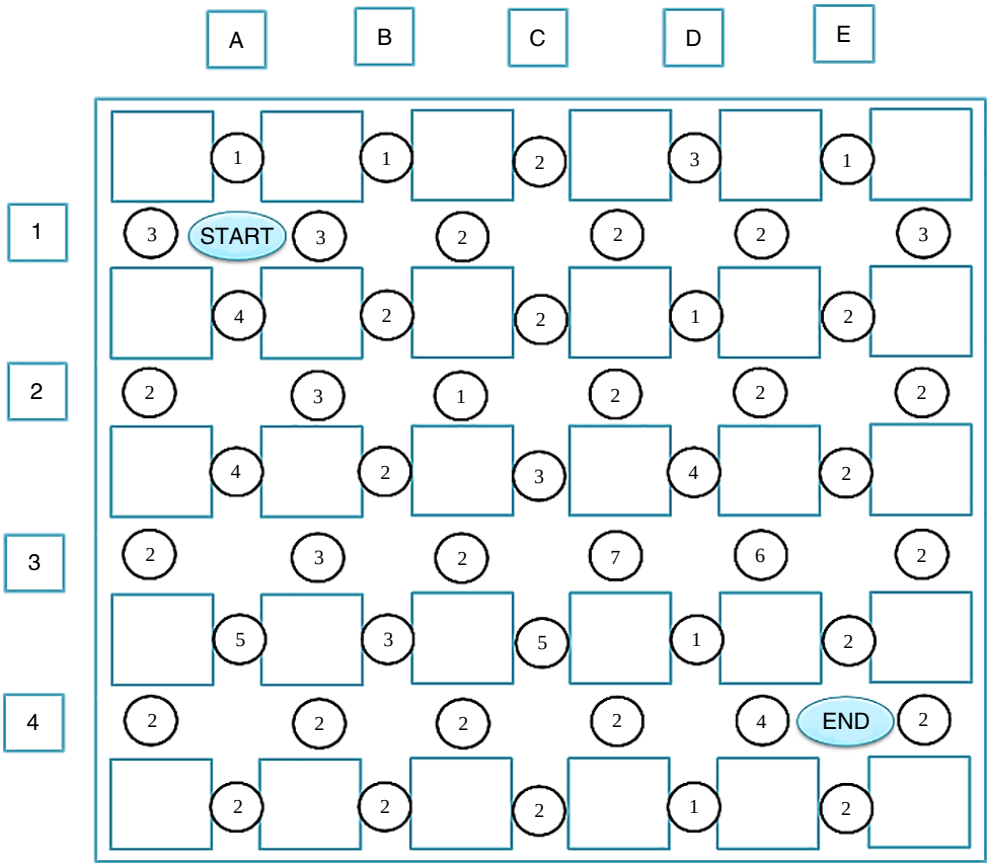


图 1-3　导航问题

现在，我准备从 START 处去往 END 处，我需要一个“智能”的软件为我导航。这个问题难吗？如果你接触过与数据结构相关的课程，那么你一定会觉得这是一个非常简单的问题——就是图论里的图遍历问题或者树遍历问题。

该导航软件预设不会给出走回头路的方案（以免进入死循环）。以 START 处为起点，我可以向上、下、左、右各走一个街区的距离：向左和向上走会到达城墙，不能再走；向右走到 B1 处，有三条路可以走；向下走到 A2 处，还有三条路可以走。所以，如果把这个问题当作一个树遍历问题，树的前两层应该非常容易构建出来。

如图 1-4 所示：X 表示走回头路或者“撞墙”，这样的路是不能走的，其下面的树结构也就不存在了；A1、B1 等交汇位置的节点则表示该处可以前往，并标注了该处之后的走法。这样一层一层往下标，最多能标出多少层呢？不清楚，要仔细算算才能知道。但可以确定的是，只需要 8 层就可以完成从 START 处（A1）到 END 处（E4）的行进，而且走法不止一种。

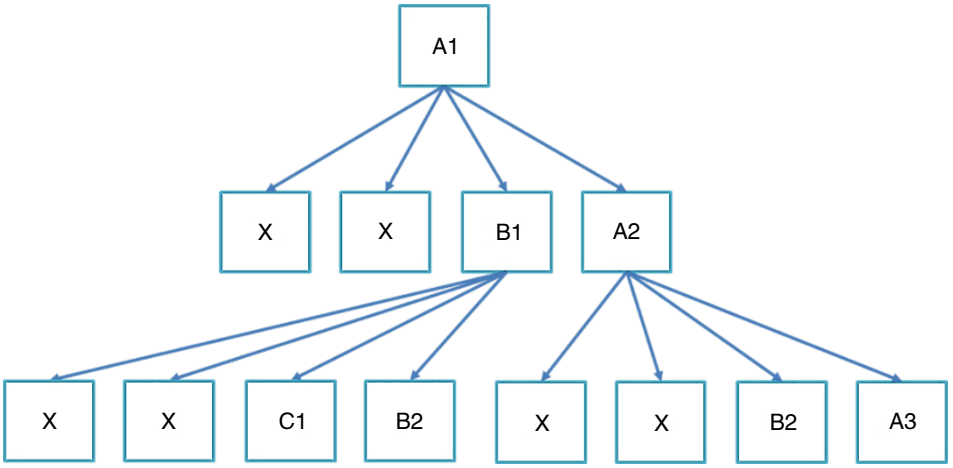


图 1-4　地图树遍历

任意列举几条路线：

A1→B1→C1→D1→E1→E2→E3→E4

A1→B1→C1→D1→D2→D3→D4→E4

A1→B1→C1→C2→C3→C4→D3→E4

……

我们不仅可以把这些路线一一列举出来，还可以把经过每条路线所花费的时间算出来（对一条路线的时间成本做一下简单的加法运算就可以了）。以路线 A1→B1→C1→D1→E1→E2→E3→E4 为例，时间成本为 $3+2+2+2+2+2+2=15$，也就是说，从 START 处（A1）到 END 处（E4），走这条路线需要 15 分钟。

如果想找到最短的路径，只要遍历路线，计算每条路线的成本，把成本最低（或者说耗时最短）的路线挑出来就可以了。这个思路再简单不过，可以保证在当前状态下是最优解——只要不出现走到半路突然发生交通事故，使得原本通畅的道路变得拥堵，或者原本拥堵的情况突然得到缓解。

现在，出现了两个问题。

第一，我们很容易用一种“智能”的方法得出这种问题的最优解，并向用户推荐这个最优解。之所以说它“智能”，是因为它确实自动实现了路程规划和推荐，挑选的是成本最低的路径。但是，在这个过程中，我没有用到任何机器学习的知识，也没有遇到任何需要通过样本学习来生成模型、确定待定系数的问题。我只用计算机专业本科一年级所学的一些基础知识就已经获解了，完全没有用到人工智能的方法——好意思叫“智能”吗？

第二，这是不是一个贯序决策问题呢？是或者不是，这也是一个问题。我想，可能会有两方面的意见。

认为这不是贯序决策问题的读者，会觉得这里面根本没有决策问题，而是纯粹的树遍历（如图 1-5 所示）问题——非常不聪明，非常不智能，这哪里这是什么贯序决策问题！

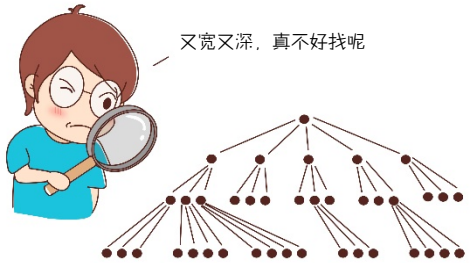


图 1-5　树遍历

认为这是贯序决策问题的读者会觉得：从起点前往终点，中途在任何一个可以选择路径的位置都进行了判断，而且每次都选择沿着最好的路线前进——是的，“最好”，没有“之一”——在每个环节都进行了选择，这难道不是贯序决策问题吗？

两方的意见，我觉得都有道理。我不是一个喜欢咬文嚼字的人，也不想拿着什么金科玉律去说教。问题就摆在这里，我们要考虑的是：用什么样的方法解决问题才是合理的。

这个导航问题，在我看来，还真可以算作贯序决策问题。第一，它“够贯序”，因为它是由一系列决策判断组合而成的一个完整的行车路径策略；第二，它“有决策”，因为在每个十字路口都有很多选择，我需要充分的理由来决定要走哪一边。这两点是事实，对不对？

或许有些读者很失望——为什么要讲这么简单的例子？我只是希望大家能通过一个简单的例子明白一个道理：数学问题不一定非要通过极为艰深的算法和技术来解决，很多时候就是“小药治大病”，如果能用简单的方法解决问题，确实没有必要摆弄那些复杂的算法。反之，为了解决一

个简单的问题而引入一大堆复杂的问题，在我看来，不是一种值得提倡的思考和处理方式。在解决具体的落地问题时，要尽可能选择复杂度可控、难度低、理论成熟的方法。像本节中这么特殊的贯序决策问题，在一定的限制条件下就会“退化”成搜索问题或遍历问题。所以，别犹豫，我们本来就应该用简单的方法去解决简单的问题。

结论已经很清晰了，刚刚这个问题就是一个贯序决策问题，只不过它是一个特例。那么，更为普适的贯序决策场景是什么样的呢？这个问题好，我们不妨想想看。

如果街区不是只有 A ~ E 这 5 条纵向的大道和 1 ~ 4 这 4 条横向的大道呢？如果有 100 条横向的大道和 100 条纵向的大道，这棵遍历树会有多“高”呢？应该最少要走 198 条路才能到达，也就是一棵 199 层的遍历树——这是非常可怕的！甚至，在很多场景中，我们很可能无法估算具体的层数。除此之外，如果我们不清楚每条路的拥堵情况，该怎么办呢？如果我们不能明确量化通过每条路预计需要多少分钟，该怎么办呢？

通过一系列科学的方法，对这类普适性问题进行体系性的求解方式和方法的归纳，才是这么多强化学习算法要解决的核心问题。

1.2.2　强化学习的建模

在 1.2.1 节中，我们对一个简单的例子进行了讨论。我只是为了说明一个问题：虽然强化学习是一种用于解决贯序决策问题的方法，但并不是所有问题都必须借助这种高深的手段才能获解。

例如，在 1.2.1 节那种层级有限且树杈上的权重（行进成本和行进时间）清晰的场景中，采用普通的树遍历方法足以解决问题。如果不知道整棵树的层级、分叉和每个树杈的权重，是没有办法用直接遍历的方法得到一个完整路径的行进策略的。

这个时候，强化学习就可以派上用场了。按照最朴素的思路，似乎应该想尽办法来构造这样一棵树，然后把问题转换成我们以前接触过的、有成熟解决方案的问题。为什么这么说呢？千万别忘了计算机是干什么用的。它能思考吗？不能。它能做的事情太单纯了，只能做加、减、乘、除运算，而且乘、除、减都是由加模拟出来的。除此之外，就是比大小。再多的处理过程，也都是加、减、乘、除和比大小这样的简单操作的组合——没办法，这是计算机天生的能力（或者说局限性）所带来的。因此，如果能够把贯序决策过程化简成树遍历过程，那么问题就回到了计算机擅长的范畴，也就容易解决了。

回想一下前面我们提到的监督学习。监督学习的方法是非常套路化的：把输入的 x 和输出的 y 之间用一个函数 $y = f(x|\theta)$ 来映射，把用来描述这种映射关系的误差描述成待定系数为 θ 的损失函数 $\text{Loss} = \sum |f(x_i|\theta) - y_i|$，通过最优化问题（Optimization Problem）的思路或方法来求解当损失函数 $\text{Loss}(\theta)$ 取极小值时 θ 的值。这就是绝大多数监督学习方法的整体思路，$f(x|\theta)$ 就是我们想要得到的模型表达式。在机器学习中，监督学习的问题在学习过程中被化简了：用函数描述误差值，并通过加和减的方式更新参数，再通过比大小的方式判断训练是否收敛的过程，就是一

个利用计算机本身的计算能力建模的过程。

当然，一个机器学习问题也有可能不被转换成求解一个函数的极小值的问题，而被转换成求解其极大值的问题。例如，待优化的函数在描述得分、匹配度、适应度等积极评价标准的时候，就是在求极大值了。

强化学习也能总结成这样的套路吗？当然能。伟大的人工智能先驱们为了我们做了非常多的铺垫，我们就沿着他们的思维脉络来逐步了解强化学习吧。

1. 主要概念和术语

我们在各种与强化学习相关的资料中都能看到类似图 1-6 的示意图，即使内容略有出入，核心意思也没有很大的差别。其中有几个对象是建模的重要因素，接下来就会介绍。不过，我想强调一下：这样建模实属不得已。为什么？回头看看 1.2.1 节中那个导航的例子。在那个例子中，我们就像在玩游戏时开启了“上帝视角”，不知不觉就启用了很多“特权”，于是，起点在哪里、终点在哪里、一共有多少条路、走每条路时需要花费的成本是多少，一目了然。所以，我们当然可以在全局范围内计算、排序，得出一个全局最优解。

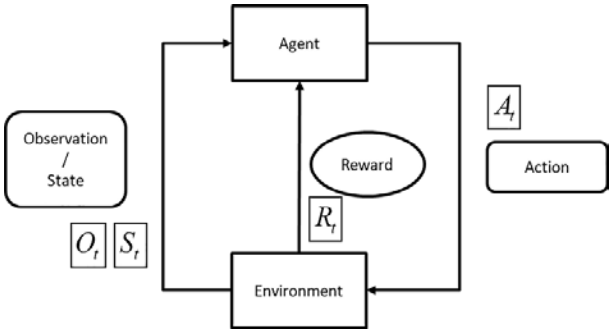


图 1-6 强化学习研究对象关系示意图

我们希望机器人能够在一个环境中自己找到那个“最优”（至少是一个“差不多优”）的决策路线，但问题是，我们并非每次都有“特权”，确切地说，我们在绝大多数情况下都不可能有“特权”。不过，一个以人类智能的思路进行训练与进化的机器人，还是要尽可能与人类的经验和推理过程保持一致，毕竟人类才是目前自然界中智慧程度最高的，否则，为什么是人类统治了地球，而不是三叶虫[①]、恐龙或者大猩猩呢？而从悲观的角度思考，强化学习成了一种无奈的选择，是在无法通过直接开启“上帝视角”解决问题的情况下采取的仿照人类探索自然环境的方法去建模的手段。那么，强化学习设定的重要研究对象都有哪些呢？

① 三叶虫是一种生活在远古时期的节肢动物，在 5 亿 ~ 4.3 亿年前发展到高峰，在距今约 2.4 亿年的二叠纪完全灭绝，在地球上生存了超过 3.2 亿年，是一类生命力极强的生物。

（1）Agent

通俗地讲，“Agent”（或者称为“Brain”、大脑、智能体）就是机器人的智能主体部分。它到底应该叫什么，其实不重要。我们只需要知道，它就是我们放到环境中去探索和学习用的机器人主体。请回想一下，在 1.2.1 节中，我们把导航这样一个具体的问题抽象成了一个树遍历问题，对不对？可是，现在的麻烦是，如果这棵树的形态不确定，那么我们不仅没有办法用普通的树遍历方法来解决问题，而且必须先构造一个用于尝试的机器人来构建完整的树（这个机器人会在这个陌生的世界里通过试探的方式来构建一棵树）。

（2）Environment

“Environment”一般翻译成“环境”，是指机器人所处的环境。对于环境，没有特别苛刻的定义，主要是指机器人所在的环境，通常我们不会深究它的具体定义。其实，Environment 就是机器人将要试探的环境，也就是客观环境（可以是一个游戏程序，也可以是其他复杂的环境）。

（3）Observation

“Observation”翻译成“观测”可能比较好。Observation 是指 Agent 能够观测（感知）到的环境信息。因为我们无法开启“上帝视角”，所以只能在非常有限的视野里观测周围的情况——是的，非常有限。这种建模方式本身也是由人类自身的认知局限性产生的。无论是人，还是机器人，能够感知的范围和内容都是有限的，因此，能感知什么，把什么当成 Observation 就好了。我们在写程序的时候，可以用一些技巧对观测的视野进行降维，也可以把在一段时间内连续观测到的情景数字化并作为当前观测到的 Observation。

（4）Action

Action（动作）是指由 Agent 发出的行为和动作，以及 Agent 与 Environment 之间发生的动作交互。Action 的内容定义很丰富，可以描述成是否做某个动作（用类似 1、0 这样的值来表示），也可以描述成一个连续的向量（例如 $[0.35, 0.21, 0.7]$），关键看机器人是什么类型的。

Action 可能（不一定）会影响机器人在下一时刻的状态，我们只要考虑人的情况就好。例如，一个人今天有没有打伞，其实不会影响明天下雨的概率，因此，在这个场景中，打伞这个动作不会对 Observation 有任何影响，该观测到下雨还是会观测到下雨。再如，在下象棋时，一个人的走棋会在很大程度上影响另一个人接下来的走棋，因此，在这个场景中，连续观测到的两个盘面会大不一样。当然，在研究的时候，肯定是前一个场景更为简单一些。

（5）Reward

“Reward”一般翻译为“奖励值”，有时也翻译为“回报值”。在本书中会混用这两种称谓，或者口语化地称之为“得分”。在强化学习中，我觉得最为有趣、最为精妙的一个研究对象兼超参数（Hyperparameter）就是 Reward。

什么是超参数呢？就是无法通过训练自动学会的参数。这与本章开头的那个监督学习的例子不同。在那个例子中，$y = f(x|\theta)$，θ 虽然是未知数，但可以通过对损失函数进行优化后求解满足误差最小的 θ 的值来求出。超参数则必须在整个训练开始之前确定，因为在训练过程中没有办法通过学习来确定这个值。

在强化学习中，Reward 是指 Agent 在做了一个动作后（或者在某一时刻）得到的用来描述其行为好坏的值（如图 1-7 所示）。如果一个行为很好，那么 Agent 可能会得到一个很大的奖励值，例如 +100、+220；如果一个行为很不好，那么 Agent 可能会得到一个很小的奖励值（甚至可能是负值），例如 –10、–500。看样子，–500 像是闯了大祸后得到的奖励值——哦，不，说“惩罚值”才更合适吧。

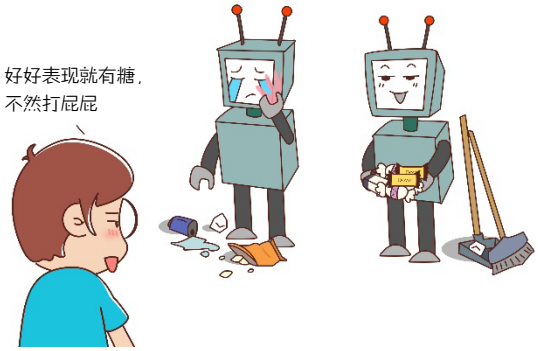


图 1-7　对机器人的奖惩

2. 观测流程

现在，我们已经基本上把强化学习领域的研究对象列全了。

从 Agent 的角度看，做一个动作 A_t，得到一个观测值 O_t，以及一个标量[①]奖励值 R_t。观测值 O_t 可以通过另一个概念来几乎等同地理解，这个概念就是状态（State），写作 S_t^e 或 S_t。机器人在一个环境中不断地试探，就会不断地得到这样一个短序列：

$$S_t \rightarrow A_t \rightarrow R_t \rightarrow S_{t+1}.$$

我知道，很多读者朋友讨厌看公式、字母、角标等容易让人头晕的东西。不过，由于在很多资料中会出现同一个概念的多种书写方式，我也只能尽量把它们都列出来。当然，我每次都会用“白话”解释一下字母或者角标的含义。例如，这些观测值 O_t 在被机器人获取后，有可能不是原原本本地被使用的——如果 O_t 有 10000 个维度，我就会保存或者关心 10000 个维度，但其中可能有很多我不需要的信息，那么，我可以考虑用任何可靠的工程手段来做一个化简的表示。

将 O_t 写作 S_t^e，e 是指特定的 Environment，t 是指某一时刻，意思就是：在某一时刻所观测

① 标量，只有大小、没有方向的值，在本书中可以直观地理解为实数。

到的状态描述信息。在这里，不必纠结于 O_t 和 S_t 的写法，因为机器人完全可以通过软件或者硬件的手段只去观测我们认为重要的状态信息，所以，直接得到 S_t 也不是难事。因此，本书不再刻意强调 O_t 和 S_t 究竟有什么差别——或许在学术上有差别，但在实践中我们不再纠结。同时，在本书中，也不区分状态 S 和 A 及时间片 t 的大小写（在其他资料中，我也看到过不少不区分大小写或者大小写混用的情况）。

其实，将短序列 $S_t \to A_t \to R_t \to S_{t+1}$ 作为观测对象来建模也实属无奈。因为这样做无法得到一棵完整的树，所以才不得不用边做动作边记录的方式来收集相应的数据。在一些资料中，会将这些短序列称为 $S \to S'$ Transition，这就是在强化学习的建模环节所能观测到的有意义的最小单位。今天的人可以从众多文、史、哲方面的资料中汲取营养，但是古代的人是没有这样的条件的。怎么办？只能通过反复实践，把这些客观观测到的现象所对应的人类行为及后果片段性地记录下来——连续记录。当积累一定量的数据之后，后世的人就能从中品味、总结，得到一些智慧的启迪了。

3. 注意事项

在本节前面的内容中，为了介绍新的概念，我给出了对应的中文和英文。在本书后面的内容中，将只使用它们的中文或者英文来表示这些概念。这些概念涉及的对象，虽然看上去简单，但仍为后面研究它们之间的复杂关系留下了伏笔。在研究正式开始之前，我要强调以下三点。

第一，下标 t 和 $t+1$ 指的是某一时刻。虽然我们研究的是连续决策问题，可是再“连续”的状态，也是由一个个小的、离散的静态状态对象构成的，Agent 只能处理这类离散的对象。也可以说，t 就是一个时间片，$t+1$ 就是下一个时间片。至于一个时间片所代表的时间究竟有多长，以及环境这个客观对象在没有收到 Agent 发出的动作 A_t 的时候，它的时间片计数是否应该继续增加，就要“具体环境具体分析”了。

第二，我们在研究强化学习领域的相关问题时，通常是站在 Agent 的角度去观察、建模、分析的，而很少会站在环境的角度去描述（除非这个环境是由我们自己构建、用来与 Agent 进行交互的）。

第三，动作 A_t、状态值 S_t、奖励值 R_t，究竟是客观存在的，还是人为设定的？这个问题很重要。

动作，很容易理解。一个机器人，无论是下棋机器人（例如 AlphaGo），还是扫地机器人，抑或自动驾驶机器人，动作的取值都是有限的。无论是离散描述的动作，例如前、后、左、右的行进方向，还是连续描述的动作，例如方向盘旋转的角度 $[-\pi, \pi]$[①]、启动或关闭一个设备，抑或它们的组合，都是在制造机器人来解决问题的时候就确定了的（如图 1-8 所示）。因此，我们基本上可以认为，动作的取值范围是客观的。

---

① 弧度单位，此处的 π 等于角度中的 180 度。

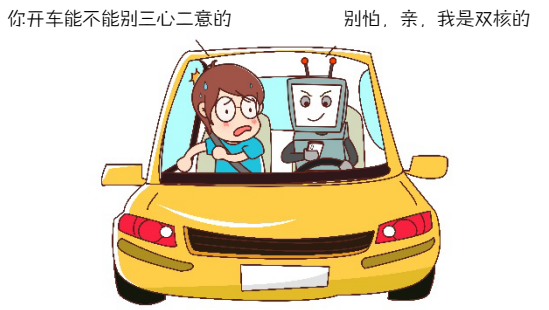


图 1-8 输出为复杂动作的机器人

状态值的数量可能比较多，因为要描述一个具体的状态，几个离散的数据可能不够用，通常需要数以万计甚至数以百万计的不同数值才能描述一个环境所能产生的状态值。它的取值基本上可以认为是主观的——因为人可以根据自己的需要对客观世界进行不同维度的刻画，所以 Agent 也可以根据自己的需要对它所感知的世界进行相应的取舍。例如，对一辆正在自动行驶的汽车，要如何描述它在某一时刻的状态呢？

即使没有做过相关的项目，也可以想一想：可能要通过 GPS 信息描述当前的位置，可能要描述当前的车速，还可能要通过各个方向的雷达测距信息来描述周围物体与车体之间的距离……粗略算下来，需要十几个维度的信息。此外，需要把车体前方和后方摄像头采集的连续的图像输入车载计算机，而图像里面的信息就多了，哪怕是一幅 800 像素 × 600 像素的三通道图像，也至少包括 144 万（$3 \times 800 \times 600$）个维度的向量……不论是可能的取值范围，还是在收集这些数据时真正观测到的时间片的数量，都是非常惊人的。而这些，都只是设想的值。至于要不要通过感知地面的坑洼程度、摩擦力的大小来改进行车的舒适程度等，完全取决于这个自动驾驶机器人是否需要解决相关的问题——可以选择感知，也可以选择不感知，没有严格的对与错，一切都以帮助机器人更好地完成任务为目标（如图 1-9 所示）。

图 1-9 机器人对状态值的感知

奖励值估计是这三个值中最特殊的一个了，它几乎完全由机器人的开发者来决定。机器人每观测到一个状态，都要尝试做一个动作（原地不动也算一个动作，你可以把它叫作“沉默”）。那么，一个动作应该获得多大的奖励值呢？环境本身是无法给出的。

奖励值看上去非常像人类自己的价值观，而人类的价值观千差万别，不仅生活在不同地区、认知水平不同的人的价值观不同，就连我们身边最亲近的人、和我们的教育背景相近的人的价值观也不同。价值观不同的主要表现，就是对同一件事情的评价标准及对矛盾的取舍原则不同。例如，孩子在公共场合犯“熊”，家长应该呵斥还是放纵？呵斥可以理解为一个负的奖励值，放纵可以理解为一个正的奖励值，这就是一种训练。训练的结果在当时可能看不出来，但在多年以后，这个孩子可能成为老师、同学、邻居都喜欢的好少年，也可能不怎么受周围的人欢迎。其实，关键就在奖励值的设定上。对一个机器人来说，在一个环境中观测到一个状态值后做一个动作所得到的奖励值，直接体现了训练者的价值观，而这个价值观将对训练结果产生直接的影响。

可以说，奖励值的不同将直接导致训练目标的不同。从另一个角度看，在同样的环境中，可以通过改变奖励值来调整训练的方向。

了解这些概念以后就可以开始训练了吗？是的，基础概念只有这么多。从第 2 章开始，我们就要进入正题了——如何通过上面这些值得到一个好的机器人策略。

1.3　本章小结

我们在研究任何问题之前，都应该划一个范围，或者确定一个大致的研究边界。然后，我们才能在这个确定的范围内进行讨论。

好了，开场的铺垫就这么多。接下来，我们就正式开始探究机器人自我进化的奥秘。

第 2 章　强化学习的脉络

虽说本书讲的是以 PyTorch 作为落地工具的深度强化学习的应用，但是，要想比较清晰地了解这个强化学习和深度学习相结合的交叉学科领域，还是要分别把这两个部分弄清楚。请各位读者暂时忘掉深度学习和 PyTorch，让我们一起仔细了解一下强化学习考虑问题的方式。

从本章开始，我们就要正式接触强化学习所涉及的各种模型、复合概念与算法了。与其说第 1 章介绍了强化学习建模时研究的对象，不如说是不得不做的对客观世界的观察手段的片段性陈述。现在，就让我们看看接下来要怎么研究、会碰到什么问题吧。

2.1　什么是策略

在第 1 章中，除了 Agent、Environment、Observation、State、Action、Reward 这些词，我还提到了一个词——策略（Policy）。我们应该如何理解机器人行为范畴内的策略呢？一般用这样一个表达式来描述：

$$a = \pi(s).$$

这里的 a 指的是动作，s 指的是状态，函数 π（跟圆周率没有半点关系）就是用来描述策略的。从形式上看，策略就像一个函数，只要输入一个状态 s，就输出一个动作 a。

如果你在其他资料中看到

$$a = \pi(s|\theta),$$

也不用紧张，这个 θ 仅仅为了说明在策略函数 π 中有参数 θ。当然，这个 θ 取什么值合适，在强化学习中是通过训练得到的。θ 的表达式通常不会太简单，不是用一两个实数就能描述清楚的。在强化学习中，可以用两种方式去理解这个函数：可以感性地理解为一些矩阵，在状态 s 以向量或者矩阵的形式输入后，与 s 进行各种矩阵计算；也可以更感性地理解为一些由 if…else… 及 for 循环组成的复杂的语句，用来对输入的 s 进行判断与处理。你觉得怎么理解更容易，就先怎么理解。

还有一种描述方式，经常写作

$$\pi(a|s) = P[A_t = a|S_t = s].$$

看上去和前面的策略写法有点像，但是略微复杂一些。这个公式表示的是输入状态 s 后，策略输出的 a 不是一个确定的值，而是一个概率分布的描述。而概率的描述，就在后面这个部分，就是 $P[A_t = a|S_t = s]$。也就是说，当输入一个状态 $S_t = s$ 的时候，应该输出的动作 $A_t = a$ 的概率为

多少。当然，就算我们不细究 0 和 1 的开闭区间的细节，这个概率也一定是一个 0 和 1 之间的数字。如果你觉得这个形式看上去不直观，让我们换一个方式来表述，如表 2-1 所示。

表 2–1　$\pi(a|s) = P[A_t = a|S_t = s]$ 的含义

	S_1	S_2	S_3	S_4	S_5	…	S_N
A_1	0.02	0.03	0	0.05	0.1	…	0.3
A_2	0.15	…	…	…	0.02	…	0.03
A_3	0.45	…	…	…	0.09	…	0.04
…	…	…	…	…	…	…	…
A_N	0.01	0.05	0.03	0.11	0.02	…	0.01

表 2-1 相当于将 $\pi(a|s) = P[A_t = a|S_t = s]$ 展开，A_1 到 A_N 表示整个动作空间 [①]中不同的动作的个数，S_1 到 S_N 表示所有可能的状态值。注意，这里的 1、2、3……N 不是时间片的次序，只用于描述状态的不同取值。现在，我们通过查表就可以得到策略的完整描述了。假如某一时刻机器人得到了 S_5 状态的输入，那么它应该输出一个动作，这个动作 a 应该有 10% 的概率输出 A_1、有 2% 的概率输出 A_2、有 9% 的概率输出 A_3……有 2% 的概率输出 A_N。大致就是这样。

我估计，可能已经有读者有疑问了。

- 问题一：策略的描述我知道了，但是策略是怎么得到的呢？或者说，怎么评价一个策略好或者不好呢？
- 问题二：既然输入状态是确定的，那么在同一个状态下，不是应该只有一个动作是最优或者最好的吗？能不能把这个最优或者最好的动作求出来，直接做这个动作，而不用每次都通过“丢一个不均匀的骰子”来决定做哪个动作呢？

这些问题都非常好。

2.2　什么样的策略是好的策略

其实，研究机器人和研究人是类似的，研究机器人的行为和研究人的行为也是类似的。在人类的世界中，评价一个事物好或者不好，通常都是基于评价者自身价值观的。不同的人对相同事物的同一属性，也会有不同的看法或者截然相反的评价。

举个非常简单的例子，如表 2-2 所示，对一顿便餐，“在家做饭”还是“外出吃饭”，不同的人会有完全不同的看法，进而导向两种完全不同的原则（或者说策略）。当然，不管怎么说，这两种方案各有利弊。

① 所有可能的动作取值。

表 2–2 对“在家做饭”和“外出吃饭”的比较

比较项目	在家做饭	外出吃饭
对食材状况的把握程度	较高	较低
食材与调料的购买成本	较高	较低
自由选择菜品的空间	较小	较大
需要额外花费的时间（例如外出）	较短	较长
对个性化口味的把握程度	较高	较低
……	……	……

我们可以尝试列出表格，在简单比较后进行判断。要知道，即便是住在同一个小区的人，做饭水平一样或者选择的是同一家餐厅（看似一切预设状态都极为相似），得出的结论也可能是不同的。例如，一个人打算吃点东西就去上班，另一个人在家闲来无事想尝尝新菜式，他们就会有不同的动作输出——一个在家做饭，另一个外出吃饭。

其实，还有很多没有列出的因素。例如，餐厅的环境、餐具的选择、服务员的态度等。这些因素不仅会影响就餐气氛，也会影响当事人对做饭成本（时间成本和材料成本）的判断。再如，有没有人一起就餐？对一起就餐的人，是吃路边摊就可以，还是需要准备一顿丰盛的家宴？这些都是不同的因素。

人类研究的策略其实是非常复杂的，不是一本书就能说明白的。不过，别慌，在机器学习的范畴里，如果想解决一个问题，通常要简单得多。不是因为机器学习本身的理论很“牛”，恰恰相反，是因为它太“弱”了——不能像人一样进行复杂的逻辑推理与判断，进行“理性+感性”的复杂分析，只能比较一个“数”的大小。什么“数”？就是我们常说的评价函数（Evaluation Function）。

评价函数的概念是什么？其实就相当于我们在上学的时候每门考试的成绩。如何评价一个人学习成绩的好坏？就看他的考试成绩如何。在同一个班里的两个同学，“明明”数学考了 59 分，“偏偏”数学考了 100 分——“偏偏”最后考上了大学，成了一名程序员。对于学生，评价函数可以作为遴选标准——谁的评价函数的值大，就选择谁。至于策略，评价函数就是“考试”，而且没有其他更好的方法。

不同的策略参加“考试”，谁的得分高，就认为谁是好策略。公平吧？得分面前人人平等。这个“考试”就是 Environment（也可以说是考题）。让不同的策略在 Environment 里面摸爬滚打，最后计算一下谁的得分高——这个思路很自然，也很顺畅。

当然，有人会问：一次考试就能说明问题吗？一次考试的得分就能客观、公正地评价一个考生的能力或实力吗？这里有没有随机的成分，有没有不公平的现象存在？这些问题也是有道理的。所以，在给持有不同策略的机器人考生出题时，我们也希望它们能有长期稳定的发挥。例如，通过多次使用策略取平均值，用统计学的方法，消除单次考试中由于题目过于简单或者过于困难而产生的对策略的评价不准确的问题。

这个话题和代码本身没有什么关系，这是一个原则，或者说一种思维方式，希望能跟大家达成共识。

至于“在同一个状态下，不是应该只有一个动作是最优或者最好的吗”这个问题，我们暂且按下不表，后面自会解答。

2.3　什么是模型

模型（Model）这个词，在不同的研究领域有着不同的释义。在强化学习领域，模型是用于预测环境中将发生什么的一套描述信息。该怎么理解呢？

对一个机器人所处的环境来说，如果这个环境是自然形成的，而不是由某个人创造的，那么，这个环境将以什么样的规律变化，通常是没办法直接表示出来的。如果要描述一个模型，可以使用如下表达式：

$$P_{ss'}^{a} = P[S_{t+1} = s' | S_t = s, A_t = a];$$
$$R_s^a = E[R_{t+1} | S_t = s, A_t = a].$$

P 是概率；竖线后面的部分是 $S_t = s$ 和 $A_t = a$，表示当某个时间片 t 的状态为 s 时，做了一个动作 a；竖线前面的部分是 $S_{t+1} = s'$，表示下一个时间片的状态为 s'。用于表示概率的 P，实际上是一个矩阵。

$P_{ss'}^{a} = P[S_{t+1} = s' | S_t = s, A_t = a]$ 的含义，如表 2-3 所示。

表 2–3　$P_{ss'}^{a} = P[S_{t+1} = s' | S_t = s, A_t = a]$ 的含义

	S_1		S_2	S_3	…	S_N
A_1	S_1	0.1	…	…	…	…
	S_2	0.05				
	S_3	0				
	…	…				
	S_N	0.12				
A_2	…		…	…	…	…
A_3	…		…	…	…	…
…	…		…	…	…	…
A_N	…		…	…	…	…

表格外层的横坐标是状态 S_t，纵坐标是 A_t。表格里层左列表示的是在其时间后方的 S_{t+1} 状态的取值，右列表示各状态取值的概率。也就是说，在外层表格中，可以通过查表得到在 S_t 状态下做一个动作 A_t 后转移到各个状态的概率。每个里层表格的形式都是这样的。想想看，在一个复杂的环境中，这个表格的“体积”应该是巨大的。

除此之外，有一个 R 的表达式，如表 2-4 所示。

表 2–4 $R_s^a = E[R_{t+1}|S_t = s, A_t = a]$ 的含义

	S_1	S_2	S_3	⋯	S_N
A_1	20	⋯	⋯	⋯	⋯
A_2	⋯	⋯	⋯	⋯	⋯
A_3	⋯	⋯	⋯	⋯	⋯
⋯	⋯	⋯	⋯	⋯	⋯
A_N	⋯	⋯	⋯	⋯	⋯

比较这两个表达式。方括号内竖线后面的部分一模一样，就不多解释了，在表格中直接对应于横坐标和纵坐标。前面的 R_{t+1}，指的是获得的奖励值，可以通过查表得到在 S_t 状态下做一个动作 A_t 会得到的奖励值。如果你在一些资料中看到这个地方写的是 R_t，也不用觉得奇怪，因为毕竟这是在 t 时间片内，在 S_t 状态下做了一个动作 A_t 所得到的奖励值——用 R_t 来表示也不能算错。所以，我们只要知道这个表达式的含义是什么就可以了，至于下标具体写哪个，我个人觉得不重要。

一旦我们掌握了这两个表达式和表格中的信息，就相当于掌握了机器人视角下的客观世界的“推背图”[①]。这样，在环境中机器人怎么做才是最好的，研究起来就比较容易了。

想想看，这样一个模型，我们要怎么做才能得到？

一种情况是，如果这个环境是由我们自己创造的，那么我们至少能看到它的源代码。这样，所有的转移概率就可以很容易地得到了，甚至可以由我们自由设定。

另一种情况是，如果这个环境是由别人创造的，我们在看不到任何源代码的情况下，其实没有什么好办法，只能一次次试探，在试探中通过统计来得到转移概率。根据大数定律[②]，只要试探的次数足够多，这个转移概率矩阵就会“稳定”下来，转移概率的值会趋近于某个数字。只不过，在这种情况下，需要的试探次数非常多，说“数以万计”一点都不夸张，尤其对那些仅状态数就上万的环境来说，试探次数简直是一个天文数字，恐怕机器人在其“一生”中都要不断地做这种试探了。

不过，话说回来，这也像人类自己一样。人类的认知范围是非常有限的——即便无数次尝试扩展观测范围，大到太阳系、银河系，小到细胞、分子、原子，人类也仍然不敢说自己穷尽了观测的范围。既然如此，在任何一个场景中，我们敢说自己观测到了所有的状态、将转移概率的统

① 《推背图》是唐太宗李世民为推算大唐国运，下令两位著名道士李淳风和袁天罡所著的一本书。这本书融合了易学、天文、诗词、谜语、图画。

② 概率论中讨论随机变量序列的算术平均值向随机变量各数学期望的算术平均值收敛的定律。在随机事件的大量重复出现中，往往呈现几乎必然的规律，这个规律就是大数定律。通俗地说，在实验不变时，多次重复进行实验，随机事件的出现频率将趋近于它的概率——偶然中包含着某种必然。

计收敛做到位了吗？我们很可能连状态的设置都做得不够充分，又怎么能得到一个理想的、完美的模型呢？

但是，我仍然觉得不必悲观。机器人的进化过程和人类的进化过程是一样的，都是在进化中不断调整和进步的——从不准确到准确，从不精确到精确。在强化学习的范畴中，你也会看到这样一个过程：只要每次的调整能够被验证为有进步，尤其是与之前的建模方式相比有进步，那么，它就可以被采纳并保留下来。虽然这种进化不是由机器人自己触发的，而是纯人工的进化，但它也是一种重要的进化。

2.4　如何得到一个好的策略

在本章前面的三节中，我们讨论了策略的含义。那么，如何判定一个策略是好的策略？什么是模型？在这一节中，我们就来给强化学习定一个基调，讨论一下得到一个好策略的原则。

当然，从“好的策略”的定义来看，比较不同的策略，就是在同样的环境中比较它们的得分，更准确地说，是比较它们得分的数学期望值（算数平均值）。哪个策略能在环境中长期稳定地得到高分（得到更多的奖励值），就是好的策略；反之，就是不好的策略。这里所说的得分，意味着以一定的前提条件来“打分”，也就是说，有既定的评分原则。

既然如此，思路就比较清晰了，通常有直接法和间接法两种思路来求解一个好的策略。

2.4.1　直接法

直接法的思路并不难理解。在策略函数 $a=\pi(s|\theta)$ 中，在输入状态 s 确定的情况下，输出的动作 a 其实只和后面的参数 θ 有关。这个 θ 就是我们在训练智能机器人的时候想要得到的最终的模型参数。

而在环境中，得分（奖励值）是什么呢？

$$r=R(a|s).$$

虽然我们可能说不清这个函数中的其他系数是什么，不过在一个确定的环境中，通常在一个确定的状态 s 下输入一个确定的动作 a，就会有得分产生。这两个函数能够结合吗？其实是可以的。

如果进行代换，可以写成这样的形式：

$$r=R(\pi(s|\theta)|s).$$

在策略 π 确定的情况下，只要取一个参数 θ 的值，然后输入一个状态 s，就会得到一个得分大小的描述。

一个策略究竟能得多少分？似乎可以通过这样的形式来计算：

$$E[R(\pi(s|\theta)|s)]=\frac{1}{N}\sum_{i=1}^{N}R(\pi(s_i|\theta)|s_i).$$

也就是把各种状态下的得分加起来，然后除以状态的数量。是这样吗？不是。虽然思路的方向没有问题，但其中的细节很重要——别忘了，状态有 N 个，可这不代表它们出现的概率是均等的。

我们用一个最简单的例子作为反例。我们训练一个机器人，让它判断出门时要不要打伞，如表 2-5 所示。我们的目的是，让机器人学会雨天出门就要打伞、晴天出门不用打伞。

表 2–5　机器人打伞得分表

	晴天	雨天
打伞	0	1
不打伞	1	0

横坐标是状态（天气），纵坐标是动作，得分信息就是 0 分或者 1 分。假如目前机器人采用的策略是：晴天状态，有 20% 的概率打伞，有 80% 的概率不打伞；雨天状态，有 90% 的概率打伞，有 10% 的概率不打伞。这个策略好不好呢？我们分别计算一下。

- 晴天状态得分：$0 \times 20\% + 1 \times 80\% = 0.8$。
- 雨天状态得分：$1 \times 90\% + 0 \times 10\% = 0.9$。

最终，整个策略的评分是多少呢？是 0.85 吗？还真不一定。如果是 0.85，就相当于把晴天和雨天的概率都设置成 50%，即 $0.8 \times 50\% + 0.9 \times 50\% = 0.85$。在这个题设的环境中，没有提到一定会这样。在两种极端情况下，即晴天的概率为 100% 或者雨天的概率为 100%，策略的评分分别是 0.8 和 0.9。

这个例子只是为了说明上面那种写法是不对的。什么样的写法才是对的？当然是求数学期望值（即加权平均值）了。可以写成

$$E[R(\pi(s|\theta)|s)] = \frac{1}{N}\sum_{i=1}^{N} R(a|s_i)\,\pi(a|s_i)P(s_i),$$

或者写成更直观的

$$E[R(\pi(s|\theta)|s)] = \frac{1}{N}\sum_{i=1}^{N} R(a|s_i)\,P(a|s_i)P(s_i).$$

对每个状态值 s_i，用它们各自的概率 $P(s_i)$，乘以策略输出的动作 a 的概率 $P(a|s_i)$，再乘以在 $R(a|s_i)$ 状态下做这个动作的评分值，最后计算它们的算术平均值，就能得到对一个策略的相对客观、公正的评分了。

然后呢？既然评分函数有了，就要想办法通过让 θ 取某个值，让整个函数取得最大值。哪怕用最“土”的方法，通过 100 个或者 1000 个不同的 θ 取值，得到 100 个或者 1000 个对应的策略，也必然会得到 100 个或者 1000 个对应的得分的数学期望值——大不了挨个儿比大小，选一个得分最高的出来。

从程序员的角度看，这个方法似乎是可行的，但仔细想想，这个方法的思路不够严谨。即使将 θ 设定为某个矩阵中的值或者一系列 if…else… 语句，也不能保证覆盖所有的策略所能取的值。既然不能保证覆盖，就不能保证这种通过比大小的方法得到的结果是最优解（这和 1.2 节中那个导航的例子完全不一样）。所以，在这种情况下，我们通常会借助一些高科技的手段，通过最优化（Optimization）方法求出 θ 取值的矩阵，或者得出构造什么样子的 if...else… 语句才能保证输出最大的奖励值的数学期望值。结论暂时放在这里，求解的细节后面我们会仔细讨论。总之，直接法通过优化策略函数 $a=\pi(s|\theta)$ 的待定系数的方法来获得好的回报值的数学期望值，这个大方向不能错。

2.4.2　间接法

策略通常用待定系数来描述。如果不方便用待定系数来直接描述一个策略，有没有其他的办法呢？也是有的，就是本节要介绍的间接法。

在某种程度上，间接法似乎比直接法更容易理解—— $a=\pi(s|\theta)$ 本身就是一种策略，这种策略实际上在做什么呢？

应该说，在做如图 2-1 所示的动作搜索与转移。也就是说，当机器人所处的状态为某个状态的时候，选择输出一个动作 a，然后迁移到下一个状态 s。那么问题来了：哪个动作 a 更好？有一个非常自然的思路：哪个状态更好，哪个动作 a 就更好。在刚刚叙述的条件下，我们可以把 θ 描述成很多 if…else… 语句的组合，判断的目标就是比较哪个状态 s 更好，从而选择其对应的动作 a。这个思路就是一个广度优先遍历树的思路，只要能够把这棵树建立起来，就可以在每一层寻找那个得分最高的分支，然后做相应的动作。可是，开启“上帝视角”的情况比较特殊，在大部分场景中是没有办法做到的，因此，对陌生的场景来说，只有当前状态 s_t、动作 a、下一个状态 s_{t+1}、获得的奖励值 r 这些可以瞬时观察到的值。能不能将瞬时奖励值 r（也就是三个分支分别对应的 r_1、r_2、r_3）作为比较对象，哪个动作 a 所对应的奖励值 r 大，就选择哪个动作呢？

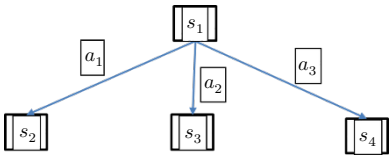


图 2-1　动作搜索与转移

好像还是有一点问题：这个瞬时奖励值的大小，其实不能真实地反映这个动作及其所对应转移到的状态的价值。因为在这个状态的后面，可能隐藏着时间更为久远、尺度更为巨大的奖励或惩罚，而这些是无法通过当时的奖励值看出来的。

说到这里，我想起了一个小故事。相传在南宋时期（1140 年），金国曾在金皇室完颜宗弼[①]（在评书里俗称“金兀术”）的带领下南下攻宋。名将刘锜[②]镇守顺昌城[③]，除了进行必要的战前动员、准备好相应的物资，他还精心设计了一系列方法来对付完颜宗弼的最强“武器”——铁浮屠和拐子马。铁浮屠和拐子马是两种重甲骑兵，对于兵种多为步兵的宋军来说，是非常难对付的。针对铁浮屠，刘锜用长枪和大斧手来对付骑手。针对拐子马，刘锜使用的方法更是令人叫绝。他命令士兵身背竹筒，冲到阵前，就在金军纳闷宋军为什么使用这种非制式装备的时候，士兵们直接打开了竹筒……煮熟的豆子从竹筒中倾倒出来，撒了一地。金军的战马酣斗多时，饥饿难忍，一看满地香喷喷的豆子，都立刻低下头吃了起来。这下子，金军的进攻节奏被破坏了，阵型也乱了。宋军趁势进攻，下砍马足，上斩敌首。金军人仰马翻，互相踩踏，折兵五千余人，战马也有大量死亡和伤残。

我觉得，通过这个故事，人和动物在智力上的差距还是高下立见的。对宋军来说，这完全不能叫“埋伏”，而是明摆着“坑人”来了——更“坑马”。然而，马就是这么天真、善良。为什么？嗯，因为它们没看过这本书，没学过强化学习。在这里讲这个故事只是为了让大家放松一下，顺便对这种瞬时奖励值的严重后果有一个认识——当前的美餐一顿，可能导致下一刻掉进万劫不复的深渊。

因此，我们通常会借助另一个函数——值函数（Value Function）来描述并期望尽可能精确地描述这个状态的价值。值函数的表达式可以这样写：

$$v_\pi(s) = E_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | S_t = s].$$

v 就是价值（Value）的意思。这个表达式的含义也比较清楚：$v_\pi(s)$ 表示，在策略 π 确定的情况下，状态 s 的值为多少（价值为多少的概率；价值越高，状态就越好）；在等号的右边是计算方法，$E_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | S_t = s]$ 指的是将下个时间片的奖励值、下下个时间片的奖励值、下下下个时间片的奖励值……相加，然后得出一个用于评估这个状态的价值的数学期望值。这个价值本身也与策略有关。例如，马看见豆子就会直接低头去吃，因此，可以想象，R_{t+1} 的值是 1，从 R_{t+2} 开始的值都是 –100（这是一个将直接导致死亡或者终身残疾的策略）。如果马无视豆子，拼死战斗，那么 R_{t+1}、R_{t+2}、R_{t+3} 等值就可能变成 10、20、–3 等，这会导向另一个结果，进而改变对当前状态的估值判断。

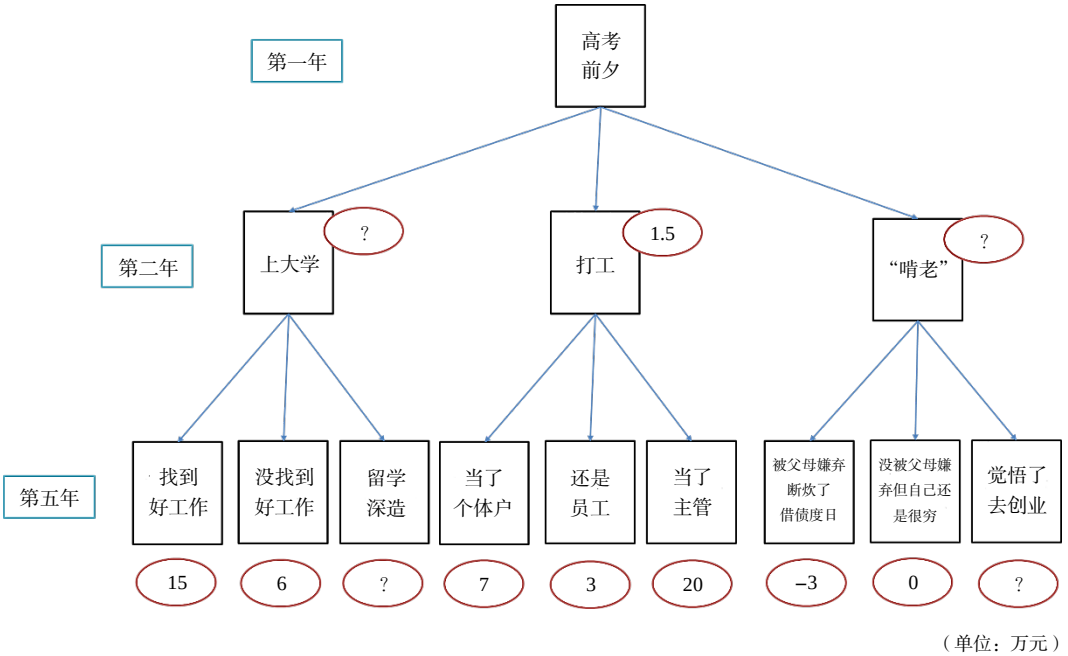
我们还是通过一个例子来具象地了解这个方法的思路。高考对中国人来说有多重要呢？高考结束后，是上大学、打工，还是“啃老”？这样的话题在社会上热议多年。然而，如何评价参加高考和上大学，如何评价参加高考和上大学的价值？还是应该通过客观、公正、科学的方法得出

① 完颜宗弼（？—1148 年 11 月 19 日），本名斡啜，又作兀术，女真族，完颜阿骨打第四子，金朝名将、开国功臣。有胆略，善射。

② 刘锜（1098 年—1162 年 2 月），德顺军（今甘肃静宁）人，南宋抗金名将。

③ 顺昌城，今天安徽阜阳一代。

结论。也就是说，既不要以自己的好恶来武断地评价，也不要仅根据一些个案来评价，而要进行大范围的统计，并在此基础上进行评价。在这里，为了评价方便，我们只用年收入这一个指标作为评分标准，如图 2-2 所示。



（单位：万元）

图 2-2　状态变化和估值评价

例如，一位普通的高中生，在高考前夕心里有了一些波动，犹豫高考之后是上大学好、打工好，还是回家“啃老”好。显而易见，打工是很快就能赚到一些钱的，哪怕是做一名普通的工人，也能有一些收入。假设一名普通工人的年收入为 1.5 万元，上大学和“啃老”在当时看来都是没有直接收入的，甚至上大学还要交学费（为了计算和讨论方便，没有把打工时产生的生活成本算在里面，只使用收入作为评分标准），而“啃老”尽管没有收入，但貌似也没有成本。在上大学和“啃老”这两项上面，我们打了问号。其实，可以把这两项的收入写成“0”（在实践中，为了计算方便，我们经常会这么做）。

几年之后，也就是我们处在下一个观测状态的时候，情况就可能会有很大的变化（生活中的情况要比图 2-2 中表现的复杂得多）。有的人找到了好工作，年薪 15 万元；有的人出国留学去了，仍然在花钱，且不挣钱；还有的人觉得打工不过瘾，自己当了小老板，也有 7 万元的年收入……林林总总，不一而足。

现在再看上大学、打工、“啃老”这三种抉择，哪一种更有价值呢？其实，就是通过人们作出相关决策之后整体（请注意，这个词也可以用“平均”来代替）有多少收入来判断的。在图 2-2 中，

可以将直线箭头理解为决策动作（决定做什么），将方框理解为状态描述。在第五年的时候，上大学的价值评价可就不是“0 收入”了，我们能够看到，后面状态的价值还是比较高的，这就解释了为什么会有一个 $R_{t+1}+\gamma R_{t+2}+\gamma^2 R_{t+3}+\cdots$ 的过程。在表达式里，我们习惯把系数 γ（希腊字母 Gamma）称作折扣系数或者折扣因子（Discount Factor），它一般是一个 0 和 1 之间的数字。显然，如果取 0，就是完全忽略在这之后状态的价值，而孤立地评估当前直接得到的回报；如果取 1，就表示不论期限有多长，都将回报记在这次“英明决策”的头上。γ 的取值也是连续变化的，表示对长期回报的重视程度：越重视长期回报，γ 的值就越靠近 1（例如 0.999、0.99 等）；越不重视长期回报，γ 的值就越靠近 0（例如 0.001 等）。这种思路也是非常自然的。

因为在机器人的“一生”中有太多的不确定因素（人又何尝不是），所以，对一个状态的评价，通常不能以一种“万寿无疆”的视野来评估，而应该采用一种重视“落袋为安”的相对短视的思维方式来评估（如图 2-3 所示）。

$$R\big(\text{上大学}\big)=R\big(\text{留学深造}\big)+\gamma R\big(\text{在大企业就职}\big)+\gamma^2 R\big(\text{找到好对象}\big)+\gamma^3 R\big(\text{生了个聪明的孩子}\big)+\cdots.$$

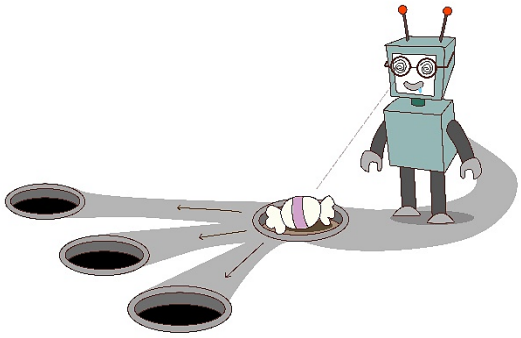


图 2-3 短视

就拿“上大学—留学深造”这条路线来说吧，可以像我在前面写的那样来评价。要知道，距离当前时间越久远的事件，对当前状态价值的影响越小。因为“上大学”对“留学深造”确实有影响，所以，把“留学深造”的价值的大部分叠加在“上大学”上是比较合理的，而“在大企业就职”“找到好对象”甚至“生了个聪明的孩子”，与“上大学”所产生的影响的关系一个比一个小，因此，估值就要打折扣了。对这个表达式，我们这样理解就可以了。

对于一个确定的策略 π 所产生的一系列状态 s 的访问轨迹及分别获取的奖励值 R 来说，对这个策略下的这个状态进行较为客观的价值评估的方法就是——多次测量取平均值。

如果你不知道要把策略 π 描述成什么样子，可以试着这么想：所谓策略，就是一个确定的原则。那么，“确定的原则”应该怎么描述呢？还是以“上大学”之后的内容为例：

```
如果大学毕业，就扔硬币
如果正面朝上，就出国留学
    如果能拿到学位
        扔硬币。如果正面朝上，就回国找工作
            如果能进入大公司
                就去大公司
            如果不能
                就去中型公司
            如果还不能
                就去读博士
        扔硬币。如果背面朝上，就在国外找工作
            ……
如果背面朝上，就在国内找工作
    ……
```

尽管用这种方式来描述这个策略显得有点“无厘头”，但它明确地表述了在碰到什么情况的时候应该做什么样的动作。这个策略不仅可以向下延展和嵌套，还可以添加更多的描述信息以处理以后 N 年内遇到的事情。按照这个原则去做抉择，一定会得到若干在这棵树上的访问路径，并获得一系列的奖励值。计算它们的算术平均值，就相当于对“上大学”这个状态的价值进行评估。现在，你应该理解间接法的值函数是如何进行估算的了吧。

我们回顾一下这种方法的思路（如图 2-4 所示）。为了使策略 π 能够在短时间内根据一个状态 s 找出应该做的最好的动作，我们希望策略 π 实现一个简单的、只有一层关系的树搜索。按照这种方式，在状态 s 下做一个动作 a 后，只要能够迁移到的任意一个状态 s' 的估值能够被正确地估算出来，那么这个人或者机器人在做决策的时候，就只需要比较估值的大小（哪个 s' 的估值大，就选择它所对应的动作 a）。这种方式操作简单，思路清晰，唯一的问题在于是否能够准确、快速地得到 s' 的估值。

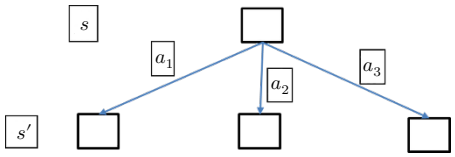


图 2-4　状态的变化

本书后面将要介绍的大部分传统强化学习算法，都会采用各种手段计算某个状态 s 的价值 $v(s)$，这也是把与传统强化学习相关的书读“薄”的一个基本思路。

2.5　马尔可夫决策过程

在本章中，不仅要介绍强化学习的脉络，还要来一点“干货”。在本节中，让我们一起了解一下马尔可夫决策过程（Markov Decision Process，MDP），如图 2-5 所示。

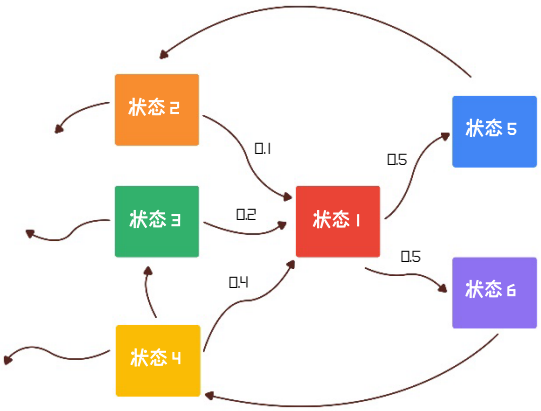


图 2-5 马尔可夫决策过程

熟悉马尔可夫[①]老爷子的读者朋友，可能一下子就能通过“望文生义”的办法猜出我下面要干什么了。可以说，马尔可夫这种“一招鲜，吃遍天”的方法，让我们在很多领域尝到了甜头，例如通信领域的编解码纠错、现代输入法的智能输入预测等。

在数学及其相关领域，提到“马尔可夫”，通常意味着：在一系列事件中，某给定事件发生的概率，只取决于前面发生的事件。言外之意就是：更久以前发生的事件不在研究范围内，只关注前面发生的事件，只针对前面发生的事件和现在发生的事件的关系来做研究。这里的“前面”，指的就是时序上的“先”。

2.5.1 状态转移

马尔可夫决策过程中的环境，需要满足条件

$$P[S_{t+1}|S_t] = P[S_{t+1}|S_1, \ldots, S_t].$$

这个等式的含义是，前一个状态 S_t 到后一个状态 S_{t+1} 的转移概率，不会随着更久以前的状态的加入而改变。如果一个环境中的状态变化满足这样的条件，就可以称状态 S_t 是马尔可夫的（或者是满足马尔可夫的）。例如，如果昨天下雨，那么今天下雨的概率是 30%，今天不下雨的概率是 70%，且不论前天是什么天气，一定会得到这个观测结果，于是，可以说这个地区的天气变化是马尔可夫的。在这种条件下，对一个策略的研究，虽然不能开启“上帝视角”，但总有一些“捷径”可以走。

假设在一个设定好的世界中，有一个机器人，它每天都会根据昨天的天气情况决定今天出门的时候带什么装备。为了讨论方便，我们把模型设置为最简单的状态，如图 2-6 所示。

① 安德雷·安德耶维齐·马尔可夫（Андрей Андреевич Марков，1856 年 6 月 14 日—1922 年 7 月 20 日），俄国数学家。

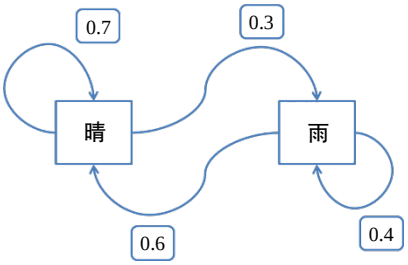


图 2-6　“原地式”天气转移概率图

在这个世界里，我们事先设计了这样的规则：如果第一天是晴天，则第二天是晴天的概率为 70%、是雨天的概率为 30%；如果第一天是雨天，则第二天是晴天的概率为 60%、是雨天的概率为 40%。现在，我们把一个机器人放到这个世界里。这个机器人每天都要出门转一圈，能做的动作只有“打伞”和“不打伞”。那么，这个机器人应该怎样设定自己的策略呢？在开始讨论之前，要补充一个重要条件，那就是奖励值，如表 2-6 所示。

表 2–6　机器人打伞奖励值表

	晴天	雨天
打伞	−0.5	1
不打伞	1	−1

在这里设定的值，与在 2.4.1 节中设定的值不一样。不过，没关系，反正这本书的内容像“系列剧”，而不像“连续剧”，场景多一点也好，不会给人留下千篇一律的印象。

事实上，在不同的环境中，这种奖励值的设定几乎是由人一手操办的。因此，本节中的 4 个奖励值和 2.4.1 节中的不同，也是有充分理由的。

- 雨天打伞，晴天不打伞，这叫正解，得到 1 分；
- 晴天原本不用打伞，但非要打伞，就会缩短伞的使用寿命，所以，得到 –0.5 分；
- 雨天直接“裸奔”搞不好会生锈的哦，所以，雨天不打伞要重重处罚，得到 –1 分。

这样就可以求出一个最棒的策略了吗？是的。其实，方法很多，尤其在简单的题设下，我们甚至可以“拍脑袋”决定。为什么呢？由图 2-6 可知，晴天后大概率是晴天，雨天后大概率还是晴天，晴天不打伞能得到 1 分。显然，“拍脑袋”决定不带伞出门是个不错的策略。事实是不是这样呢？我们往后看。

2.5.2　策略与评价

“原地式”转移图（图 2-6）不够直观，让我们换一种直观的方式，如图 2-7 所示。

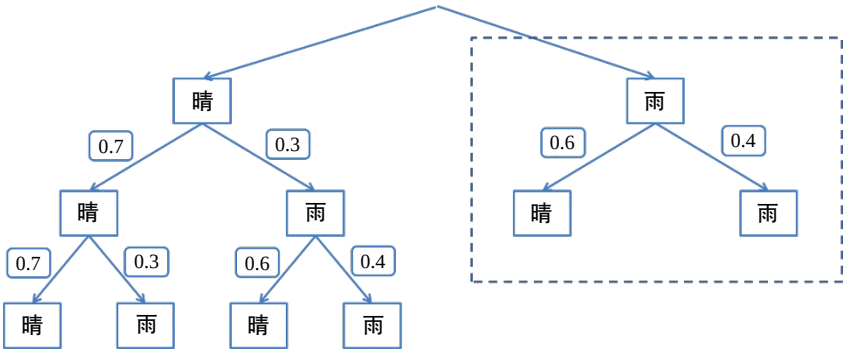


图 2-7 “时序式”天气转移概率图

是不是清楚多了？从上到下，一层就代表一天，时间一天一天地流逝，每一天“晴”和“雨”的概率都根据前一天天气的不同而不同。圆角方框里的数字是概率（例如，晴天之后的一天是晴天的概率为 0.7，晴天之后的一天是雨天的概率为 0.3）。右侧虚线框内的部分，其实和左侧右下角三个实线框组成的部分是一样的，它只是时间上的一个分支。现在我们来看左侧的两层分支，只要能把左边的那个分支研究清楚，就能把右边的那个分支研究清楚。

由于这个模型是马尔可夫的，机器人出门是否打伞根本不会影响转移的概率变化。那么，在每两层（也就是每两天）中间添加一个对动作选择的表述，会怎么样呢？假设机器人看到某一天是晴天，第二天会有 50% 的概率选择打伞出门，有 50% 的概率选择不打伞出门，应该是什么样子呢？如图 2-8 所示。

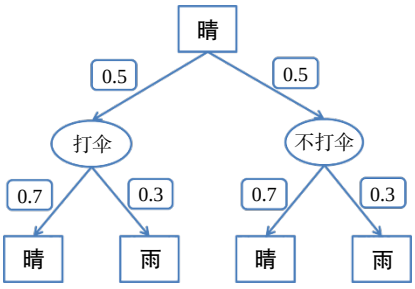


图 2-8 带有动作的天气转移概率图（观测到晴天）

无论在晴天时做什么动作，第二天“晴”和“雨”的概率都是“三七开”，因此，第二层圆角方框中的值一定是 0.7、0.3、0.7、0.3。打伞和不打伞的概率各为多少呢？这就是策略。这个例子中的策略（让机器人采用晴天后的一天打伞和不打伞各 50% 的概率），写出来应该是这样的：

$$0.5 = \pi\big(a = \text{打伞} \big| s = \text{晴}\big);$$

$$0.5 = \pi\big(a = \text{不打伞} \big| s = \text{晴}\big).$$

s = 晴，指的是前一天；a = 打伞，指的是今天。

别忘了前面的假设：机器人只能根据前一天的观测结果来决定今天的动作策略。按照这种策略，晴天时的奖励值是多少呢？如表 2-7 所示。

表 2–7　晴天时的奖励值分布

动　　作	动作概率	天　　气	天气概率	奖 励 值	加权平均奖励值
打伞	0.5	晴	0.7	−0.5	−0.175
打伞	0.5	雨	0.3	1	0.15
不打伞	0.5	晴	0.7	1	0.35
不打伞	0.5	雨	0.3	−1	0.15
完整策略	—	—	—	—	0.175

在上述情况下，晴天的完整策略所得到回报的数学期望值为 0.175。也可以说，当前一天为晴天时，按照 50% 的概率打伞、50% 的概率不打伞的策略估算，得到的平均回报值为 0.175。这也属于典型的“现世报”或者说瞬时奖励值，即没有添加任何远期回报值的评估，通过相乘和相加就能直接得到结果。

在同样的情况下，假设观测到雨天后，第二天也是 50% 的概率打伞、50% 的概率不打伞，会得到多大的回报值呢？如图 2-9 所示。

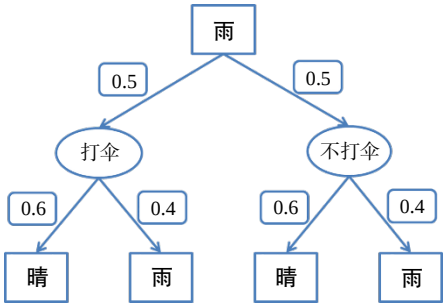


图 2-9　带有动作的天气转移概率图（观测到雨天）

采取的策略是：

$$0.5 = \pi\left(a = \text{打伞} \middle| s = \text{雨}\right);$$

$$0.5 = \pi\left(a = \text{不打伞} \middle| s = \text{雨}\right).$$

如表 2-8 所示，很容易就能计算出来，这种情况下的奖励值为 0.15（比观测到晴天时的奖励值小一些）。对此进行简单的定性分析：因为晴天之后是晴天的概率比雨天之后是晴天的概率大，所以，以同等概率打伞出门的机器人会得到较小的奖励值。

表 2–8 雨天时的奖励值分布

动 作	动作概率	天 气	天气概率	奖 励 值	加权平均奖励值
打伞	0.5	晴	0.6	−0.5	−0.15
打伞	0.5	雨	0.4	1	0.2
不打伞	0.5	晴	0.6	1	0.3
不打伞	0.5	雨	0.4	−1	−0.2
完整策略	—	—	—	—	0.15

可以明确的是，我们已经“做”出了一个策略——不是训练出来的——就是策略 π。

$$0.5 = \pi\big(a = \text{打伞} \big| s = \text{晴}\big);$$

$$0.5 = \pi\big(a = \text{不打伞} \big| s = \text{晴}\big);$$

$$0.5 = \pi\big(a = \text{打伞} \big| s = \text{雨}\big);$$

$$0.5 = \pi\big(a = \text{不打伞} \big| s = \text{雨}\big).$$

这“四句话”组合起来，就是一个完整的策略。如果把它们写成程序，应该是一个类似“查表+if…else…语句+随机数”的简单策略。即便只靠“脑补”，相信大家也能把这个程序琢磨得差不多。现在，我们能不能量化评价这个策略的水平或者能力呢？可能暂时无法给出完整的评价。毕竟一个状态下的“现世报”的奖励值太片面了，要想对某些值进行相对准确的预估，需要将眼光放长远些，而眼下，我们只有“眼下”的内容……不要紧，我们至少可以先写出基于一个策略的某个状态的价值的表达式：

$$v_\pi(s) = \sum_{a \in A} \pi(a|s) q_\pi(s, a).$$

从广义上说，策略就得用这种看上去既抽象又枯燥的东西来表示，因为越具象的东西，所能指代的内容就越具体，迁移性也就越低。上面这个表达式说的就是：在策略 π 驱使下的状态 s 的价值，可以通过等号右边的部分计算出来；等号右边的部分，表示把所有的动作以策略所规定的输出概率 $\pi(a|s)$ 取一遍，然后分别乘以各自的估值 $q_\pi(s, a)$。

这里引入了一个新概念，就是 $q_\pi(s, a)$。

$q_\pi(s, a)$ 表示，在策略 π 的驱使下，以状态 s 做动作 a 的估值（这个估值就是价值估值）。在强化学习中，q 的含义是没有二义性的，就是指动作的估值（与 $v_\pi(s)$ 的视角不同，$v_\pi(s)$ 是状态的估值）。这样，在本节的例子里，晴天这个状态在策略 π 下的价值为：

$$v_\pi\big(\text{晴}\big) = 0.5 \times q_\pi\big(\text{晴}, \text{打伞}\big) + 0.5 \times q_\pi\big(\text{晴}, \text{不打伞}\big).$$

别管这个数字具体是多少，先弄明白它们之间的数量关系。这就是前面我们通过加权平均的

方法算出来的价值，只不过，在前面算出来的是“现世报”的价值。

$q_\pi(s,a)$ 应该怎么计算呢?

$$q_\pi(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s').$$

公式就是这么冷冰冰的，我们还是画一幅图来表示吧（如图 2-10 所示）。

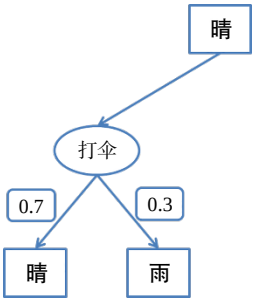


图 2-10　q_π(晴, 打伞)

假如，要评价 q_π(晴, 打伞) 的估值，应该怎么计算呢?

根据图 2-10，“打伞”这件事已经发生了，“晴”和“雨”的概率是“三七开”。先是“现世报”部分 R_s^a，也就是“晴天概率 x 晴天奖励值+雨天概率 x 雨天奖励值”（$0.7 \times 1 + 0.3 \times (-1)$）。接下来，是一个打了折扣的远期估值。在这里，我顺手给出了 $\gamma = 0.9$，也就是到达每个状态 s' 的概率分别乘以其状态估值的这样一个求和的数学期望值，具体写出来就是：

$$q_\pi\big(\text{晴}, \text{打伞}\big) = \big(0.7 \times 1 + 0.3 \times (-1)\big) + 0.9 \times \Big(0.7 \times v_\pi\big(\text{晴}\big) + 0.3 \times v_\pi\big(\text{雨}\big)\Big).$$

q_π(晴, 不打伞) 也能通过这种方式得出具体的表达式。不过，写出这个表达式以后，你心里可能一下就凉了——这简直成了循环论证，v_π 里面套着 q_π，q_π 里面套着 v_π，没完没了。其实，可以写成 v_π，也可以写成 q_π，最好能通过传统的解方程的方法，用 q_π 来代换 v_π——只剩一个未知数，解起来应该容易一些吧。

如果能够解出 $q_\pi(s,a)$，并且能够得到一个最优化的 $q_\pi(s,a)$，就得到了一个完整的策略。从形式上看，$q_\pi(s,a)$ 直接表达了在状态 s 下做动作 a 的价值有多大，这对机器人的自动决策来说就是查表或者做选择题，非常直观。这个表达式是这样的：

$$q_\pi(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in A} \pi(a'|s') q_\pi(s',a').$$

这个公式实在太长了，令人不忍直视，但实际上，它只是将 v_π 做了代换。只要你能理解 v_π 和 q_π 的含义，那么这个公式里就没有新概念了，只有旧概念的堆叠。更让人沮丧的是，这个方程很难通过传统解方程的方法解出来。一般来说，一个好的 π，需要通过值迭代、策略迭代（或者

我们将在第 5 章中学习的时间差分法）等方法找到。如果你现在还不习惯，后面还有很多机会来熟悉它们。

2.5.3 策略优化

有关策略优化，在本节中只介绍策略迭代（Policy Iteration）。

策略迭代操作起来比较简单，大致过程如下。

①以某种策略开始在环境中做动作，就像我们试着执行那个以“五五开”的概率打伞的策略一样。在这个时候，会得到相应的 $v_\pi(s)$，当然，也可以得到相应的 $q_\pi(s,a)$。

②在一个确定的策略下，得到 $q_\pi(s,a)$ 的值。只要能得到这个值，就一定能在确定的状态 s 的输入下，从 $q_\pi(s,a)$ 中找到估值最大的 $q_\pi(s,a)$ 的值。我们还是通过机器人根据天气决定是否打伞的例子来讲解。有两个不同的状态，以及两个不同的动作，所以，表中一共有 4 个值。如果有 n 个不同的状态，以及 m 个不同的动作，就应该有一个 $n \times m$ 的表来保存 $q_\pi(s,a)$ 的值，同时借助这个表输出一个完整的策略——每一次都挑选估值最大的动作作为新的策略，然后再来一次。我们将这个策略命名为 π'。

③使用 π' 在环境里做动作，会得到新的 $v'_\pi(s)$ 和 $q'_\pi(s,a)$，而且比前一个策略 π 要好（因为每次都会选择比原来的估值大的动作前往估值大的状态）。一个迭代结束后，还会产生 π''、π''' 这样的策略。只要将这个过程持续下去，总会收敛到最优策略上。

这就是策略迭代的方法，看上去有点像“贪心法”，因为每次都选择得分最高的动作，每一代都比上一代进步一点。最优的策略肯定是存在的，因为机器人表现得再好，也是在每个状态下得到一个有限的分数，做得好分数就高一些，做得不好分数就低一些，而策略估值的上限和下限都是存在的。

2.6 Model-Based 和 Model-Free

在学习强化学习的过程中，有两个名词早晚会出现在我们面前，就是 Model-Based 和 Model-Free。在一些资料中，我们经常会见到“这是一个 Model-Based 的算法”或者“这个方法是典型的 Model-Free 的算法”的说法。“Model-Based”通常被翻译成“基于模型”，“Model-Free”通常被翻译成“无模型”。

可能有人会问：为什么会有这样两个算法呢？这就要从它们各自的含义说起。

2.6.1 Model-Based

“Model-Based”既然被翻译成“基于模型”，那么关键就在于理解什么是模型。这里的“模型”，是我们常说的用机器学习的方法训练出来的模型吗？不是。这里的模型是指，在一个环境中各个状态之间转换的概率分布描述。还是以天气为例。我在广东省珠海市，想要用刚刚提到的“模

型”的概念描述天气（或者说建立一个天气模型），应该怎么做呢？我要想办法建立一个表格，如表 2-9 所示。

表 2–9　天气模型

	晴	阴	多云	有雨
晴	0.3	0.1	0.3	0.4
阴	0.4	0.1	0.1	0.4
多云	0.2	0.2	0.3	0.3
有雨	0.6	0.1	0.2	0.1

左侧的纵坐标表示第一天的天气状况，上方的横坐标表示第二天的天气状况。经过长期的统计，得到了这样一个天气转移概率表。其中，第 3 行第 5 列表示在第一天是阴天的情况下第二天有雨的概率。

这就是一个天气模型，和我们在本章开头讨论的 $S \to A \to R \to S'$ 相比，少了 A 和 R。拥有了这样的模型（Model），我们能做什么呢？不管是否满足马尔可夫特性，只要有这么一个模型存在，就必然能够在初始状态确定的情况下找出一条满足特定要求的路径。只要模型是确定的，转移概率就是确定的。只要转移概率是确定的，我们就能知道对应于 $S \to S'$ 的转移概率，以及在状态 S 下做什么样的动作 A 会有最高的回报值。这样，一个状态的估值 $v_\pi(s)$，以及一个状态下的动作估值 $q_\pi(s,a)$，就比较容易被准确地估计出来了。

2.6.2　规划问题

在 1.2.1 节中，我们提到过一个在四四方方的城市范围内导航的例子，它就是一个比较典型的规划问题的例子。

规划问题（Planning Problem）是运筹学的一个分支，是用于解决决策问题的，或者说，是用于在一定的约束条件下得出最优决策的。规划问题常用的套路，要么是解方程组，要么是解不等式组，并试着求出满足限制条件的极大值或者极小值。1.2.1 节中那个导航的例子比较理想化，由于解的空间比较小，我们采用了最为直接的穷举排序的方式来求出满足路程成本极小值的路径。对于解的空间较大，尤其是解的空间是连续值的情况，通常就要使用其他解法了，例如梯度下降（上升）法、模拟退火算法、遗传算法等（求解一个函数处于极小值时的函数自变量）。

在 1.2.1 节那个导航的例子中我们曾经说过，由于开启了“上帝视角”，用 GPS 等高科技手段掌握了全城的道路情况，在求解这个问题时，我们根本不需要建立一个含有待定系数的模型、通过正向传播让它产生误差、最终通过最优化的手段把极小值解出来的过程——在这里，完全没有使用强化学习的必要。

这就是规划问题的应用场景了。如果在一个环境中，能够得到在一个状态下做某个动作后转

移到另一个状态的概率，并且清楚地知道在这个过程中获得的奖励值的数学期望值，那么，这个问题就是一个规划问题。也就是说，可以通过遍历树、解方程、解不等式等一列工程方法求一个函数的极小值来获解，不需要采用机器学习的手段，更不需要采用强化学习的手段。我想，编过代码、写过遍历树的程序员朋友，一定明白我在说什么。

2.6.3 Model–Free

理解了 Model-Base 的概念，就可以理解 Model-Free 的概念了——Model-Base 的对立面。

在前面提到过，如果一个问题看起来满足规划问题的条件，那就把它当成规划问题来解决，不需要使用强化学习算法。可是，模型的建立工作由谁来完成呢？如果在环境建立之后没有人去建立模型，我们就不能进行人工智能手段上的策略探索和学习了吗？答案是否定的。事实上，情况恰恰相反：有人帮我们做好模型的情况很少出现。在绝大多数场景中，模型是未知的，至少是未曾精确量化的。怎么办？其实，这才是强化学习希望得到的更具普适性的方式和方法。

随着学习的深入，你会了解更多的 Model-Free 算法。现在，你可以先建立一个基本的印象。例如，Q-Learning 是通过不断求解一个状态下的动作估值函数 $Q(s,a)$ 来进行策略学习的，它并没有采用先根据统计结果做出一个模型再做规划的方法，而是直接以类似查表的方法，估算 $Q(s,a)$ 中每个“小格子”的值，从而进行建模和求解的。这个思路是很好的——我们不是“先知”，怎么知道模型长什么样？因此，采用一个直观的方法来解决问题会更靠谱。

2.7 本章小结

在这一章中，我们一起整理了强化学习的知识脉络。

强化学习是机器学习的一个分支，要想把它弄明白，需要有扎实的数学和机器学习基础理论功底。然而，数学和机器学习功夫过硬的读者，已经能够去啃大部头的经典教材了。本书希望服务的读者，主要是对强化学习感兴趣，但掌握的知识比较有限的读者，因此，只能在有限的篇幅内进行适当的取舍，以强化学习为主线，串联讲解必须掌握的知识点，并通过前后呼应的方式让读者跟随本书的内容反复学习。只有这样，才能尽量避免出现“零零散散地阅读了一大堆概念，但在脑子里还是无法将它们联系起来”的问题。

现在，我们碰到的问题是什么？做出一个能在一定的范围和环境中智能地行动的机器人。什么叫“智能”？能够高质量地完成一个完整的任务就叫“智能”。

因为一个完整的任务是由多个步骤或状态的转移构成的，所以，在建模时，是以 $S \to A \to R \to S'$ 的方式来描述的。

如果任务比较简单、已知的环境条件描述比较充足，或者说，任务可以用传统的解决规划问题的方法及穷举、遍历的方法解决，那么，就直接使用这些方法来解决。

如果直接通过上述方法无法完成任务，那么，先看看它是不是 MDP，或者看看它是否有完整

的 Model 描述。如果有，那就太好了，这仍旧是一个规划问题，可以采取规划问题的解决方案来解决。通过基于 Model 的状态转移来统计并计算当前哪个状态下的哪个路径是期望的奖励值最高的——这就是 Model-Base 的方式。如果 Model 信息不全，或者不方便通过推导 Model 来建立整个模型，就需要借助其他方式。例如，通过 Model-Free 的方式，一边在环境中试探，收集像 $S \rightarrow A \rightarrow R \rightarrow S'$ 这样的 Transition，一边通过统计的方法更新一个状态下的不同动作的估值。当某个状态下的估值逐渐变得准确的时候，就可以通过类似查表的方法得到一个确定的、优质的动作决策了。

这是一个把解决问题方法的复杂度提升的过程。如果能使用简单的方法，就尽量使用简单的方法。实在不行，才退而求其次，用基于模型的统计方法进行规划，逐步得到每个状态下的最优动作。如果还是不行（例如没有模型信息），就只能一边试探，一边计算估值，通过大量的实验，把空间“试探”出来，把一棵完整的搜索树“强化”出来。这就是解决相关贯序决策问题的完整思路。

我不想咬文嚼字，从学术的视角去解释概念或者抛出枯燥的术语，那样做对理解问题帮助不大。我认为，顺着一个自然的推导过程、采用一种具有生发性的思维去考虑问题，应该会对理解问题的全貌有更大的帮助。

接下来，我们将紧扣 Model-Free 思想下的算法族和算法体系进行讨论，这些都是理解强化学习思维的重点。读到这里，希望你能感觉到，这不是一本照本宣科的教条手册，而是一个能与你一起聊天、一起思考的伙伴。

第 3 章　动态规划

为了准确地估算一个状态的价值 $v(s)$，或者在一个状态下的某个动作的价值 $q(s,a)$，人们还是想了不少办法的。在第 2 章中，讲完马尔可夫决策过程之后，我直接介绍了如何用策略迭代的方法来实现这个目标。

在从这一章开始的连续三章中，会分别介绍一种用于准确估算一个状态的价值 $v_\pi(s)$ 或者在一个状态下做某个动作的价值 $q_\pi(s,a)$ 的思维方法。因为在所有基于状态估值的方法中，只要能够计算每个状态下每个动作的估值，就可以得到一个完整的策略描述。虽然这三种思维方法是在同一个环境中通过不断做动作来收集 $S \to A \to R \to S'$ 的，但处理思路略有不同，因此，它们各自的优缺点也不尽相同。在这一章中，我们先来了解一下动态规划（Dynamic Programming，DP），也有人称它为值迭代（Value Iteration）。

动态规划是一门相对独立的学科，是运筹学的一个分支学科。运筹学本身的学科体系就很复杂，而且在不断地完善和壮大，包括线性规划、非线性规划、整数规划、组合规划、图论、网络流、决策分析、排队论、可靠性数学理论、库存论、博弈论、搜索论、模拟等分支。动态规划关注的是如何用数学方法求解一个决策过程最优化的问题。20 世纪 50 年代，美国著名的数学家理查德·贝尔曼 [①]提出了针对多阶段决策过程问题优化的思路，并创立了解决这类过程优化问题的新方法——动态规划。1957 年，理查德·贝尔曼出版了 *Dynamic Programming* 一书，堪称该领域的开山之作。

3.1　状态估值

说到动态规划，我觉得，没必要纠结它“不动态”时的样子。简单地说，动态规划是一种建模和解题的思路。在这种思路的指导下，原始的问题（大的、复杂的问题）会被分解成多个可解的且结果可保存的子问题。一旦所有的子问题获解，原始的问题就获解了。

解决这个问题的思路，还是要从第 2 章中多次提及的表达式 $v(s)$ 和 $q(s,a)$ 开始说起。因为 $v(s)$ 可以用 $q(s,a)$ 表示，$q(s,a)$ 也可以用 $v(s)$ 表示，所以，我们尝试用 $q(s,a)$ 来代换 $v(s)$，写出表达式

$$q_\pi(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in A} \pi(a'|s') q_\pi(s',a') \,.$$

① 理查德·贝尔曼（Richard Ernest Bellman），美国数学家，动态规划理论的创始人。本书中经常提到的贝尔曼方程因他而得名。

这种视角是合理的，并给出了优化方案。现在，反过来，把 $q(s,a)$ 表示成 $v(s)$，会得到什么呢？应该会得到表达式

$$v_\pi(s)=\sum_{a\in A}\pi(a|s)(R_s^a+\gamma\sum_{s'\in S}P_{ss'}^a v_\pi(s')).$$

第 2 章中机器人根据天气决定是否打伞的例子，也可以写成更为广义的由字母和数字角标组成的表达式。

对于一个状态的估值 $v(s)$ 来说，可以寻求一种类似表达式 $q_\pi(s,a)$ 的思路，如图 3-1 所示。

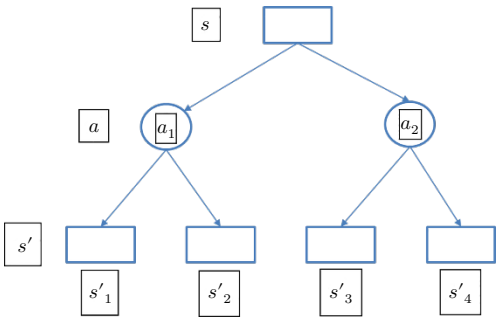


图 3-1　Transition 的树结构

为了加深印象，这次我们用字母和数字角标来表示。当我们看到一个状态空间是一棵较为完整的树的时候——准确地说，是一棵完整的树的一个枝杈——在状态 s 下，动作空间 a 中有 a_1 和 a_2 两个动作，在做 a_1 和 a_2 这两个动作的时候，会分别转移到 s'_1、s'_2 及 s'_3、s'_4。那么，根据状态估值的定义，$v(s)$ 应该等于什么呢？应该是某种加权平均值。什么是加权平均值？就是这 4 个状态各自的概率和这 4 个状态各自估值的积的平均值。打个比方，我有 10000 元，用其中的 10% 的投资了一个收益率为 5% 的理财产品，用其中的 40% 投资了一个收益率为 10% 的理财产品，分别用其中的 25% 投资了收益率为 8% 和 6% 的理财产品，现在要评价我的投资盈利金额是多少。计算这个组合策略的收益：

$$R=\frac{10000\times(10\%\times(1+5\%)+40\%\times(1+10\%)+25\%\times(1+8\%)+25\%\times(1+6\%)}{10000}=1.08.$$

这个投资组合的收益率为 8%。

在这棵树上，计算过程几乎是一样的。前面这个公式的一个具象一些的写法是这样的：

$$v_\pi(s)=\sum_{a\in A}\pi(a|s)(R_s^a+\gamma\sum_{s'\in S}P_{ss'}^a v_\pi(s')).$$

公式写出来虽然简洁，但可读性降低了。在这里详细解释一下各个部分。$v_\pi(s)$ 是顶端节点的价值估值，也就是这个投资组合的最终收益。$\sum_{a\in A}\pi(a|s)$ 表示按照策略 $\pi(a|s)$，输入 s 后输出动作 a 的概率，把这个概率和 $(R_s^a+\gamma\sum_{s\in S}P_{ss'}^a v(s'))$ 部分相乘，然后做求和运算。后面这个部分的

意义是，在 $\pi(a|s)$ 做动作时可以立刻得到的奖励值，加一个折扣系数 γ 与 $\sum_{s\in S}P^a_{ss'}v(s')$ 的积，也就是 $\sum_{s\in S}P^a_{ss'}v(s')$ 转移到其中一个状态 s' 的概率和该状态估值 $v(s')$ 的积——相当于用投资金额乘以相应的理财产品的收益率。整个公式的计算过程是一个二重嵌套的循环：遍历这棵树上的每一个分支，不管是 1 个动作，还是 100 个动作，不管是有 100 个不同的状态 s'，还是有 10000 个不同的状态 s'，总而言之，相当于一个加权平均的过程。平均计算在这个公式里面没有直接体现，因为它已经包含在概率计算里面了。

这样一来，求解 $v_\pi(s)$ 就依赖于 $\pi(a|s)$、R^a_s、$P^a_{ss'}$ 和 $v(s')$ 的准确值了。这意味着什么呢？聪明的读者朋友仔细想一下，很快就能得出结论。在这里，以下两件事情是明确的。

- 如果每一层都能够通过这样的方式逐级估算，这棵树不管有多高，都可以一层一层向上估算，直到把顶层的任何一个节点的 $v_\pi(s)$ 估值都计算准确——大不了多花些时间，方法肯定是可行的。
- 毫无疑问，这个方法严重依赖 Model（至少应该是一个满足 MDP 的对象）。如果 $\pi(a|s)$、R^a_s、$P^a_{ss'}$ 和 $v(s')$ 里面有任何一个不能直接使用这个方法，就没有办法进行计算。所以，这种方法的局限性很大，对条件的要求也比较苛刻。

这种方法的形式非常简单，也很容易理解，但是对条件的要求比较苛刻。也就是说，很多不满足 MDP 的模型根本不能使用这种方法。可能有人要说：“我可以让机器人在环境里不断自己尝试啊。尝试得足够多了，是不是就能通过统计得到 R^a_s、$P^a_{ss'}$和 $v(s')$ 了？而且，对策略 $\pi(a|s)$，也可以人为确定，或者随机做一个策略，通过动态规划来评估这个策略在某个状态下的价值啊。”这个方法不能说是错的，不过，它不符合动态规划的使用原则。而且，一旦使用这种方法，那就不是动态规划了，而是另一种方法——先卖个关子，在后面会提到。

3.2 策略优化

使用动态规划（或者说值迭代）的方法，怎么做优化呢？方法也是非常简单的，有下面 3 个步骤。

①以一种策略 π 开始在环境中做动作，这个时候，就会得到相应的 $v_\pi(s)$ 和 $q_\pi(s,a)$。这个步骤与策略迭代的第一步没什么区别。

②因为在时序上属于树遍历问题，所以，要想准确估算树上部的状态节点的值，就要先估算树下部的状态节点的值。树下部的状态节点的值容易估算吗？节点位置越靠下，邻近时间终点的位置越近，计算的代价就越小（节点位置越靠上，就表示距离开始的时间越长，在计算估值的时候要参考的时间就更长一些）。从下向上，逐层估算 $v_\pi(s)$，用类似递归的方式向上传递。表达式写出来是这样的：

$$v_{\pi-\text{new}}(s)=\max_{a\in A}(R^a_s+\gamma\sum_{s'\in S}P^a_{ss'}v_{\pi-\text{old}}(s')).$$

$R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_{\pi-\text{old}}(s')$ 表示，一个由动作 a 引发的“现世报”的奖励值加上它达到各个状态的概率和各个状态的估值——仍然可以通过本章开头提到的那个投资组合的例子来理解。

$\max\limits_{a \in A}$ 的意思是，在众多不同的 a 里，只取那个能保证后面的外层括号里的表达式的值最大的 a。也就是说，如果有一个动作“力压群雄”，比其他的动作都要好，那么，上一层的 s 的估值，就可以考虑只做这一个动作，并以这个前提进行迭代更新。道理很容易理解：如果知道有一个动作比其他动作都要好，为什么还要做其他动作？这就是这个表达式的基本含义。

③持续以这个逻辑进行更新，$v_\pi(s)$ 就会不断地变化。直到最后，$v_\pi(s)$ 基本上不再变化了，就停止更新。此时的策略，就是每次都选择能够导致状态迁移到最有价值状态的动作 a。

3.3　本章小结

在本章中，我们一起认识了动态规划。可以看出，这种方法的局限性还是非常明显的。我们已经反复强调，动态规划要求研究对象满足 MDP，因此，在实际工作中，这种方法的应用并不广泛，我在编写程序的时候，一般也不使用动态规划的方法计算估值。

作为一种非常典型的针对状态估值进行计算的方法，动态规划的思路是非常容易理解的。所以，将动态规划作为状态估值和策略优化的入门方法来学习，也是非常合适的。通过类似于递归求解的方式，逐层估算，如果每一层的估算都是准确的，就能迭代向上传递，把上面各层的各个状态的值估算准确。只要按照这个规则，就能把每个状态的值估算准确——这是一个有“远见”的估值。因此，用简单的树广度优先遍历的方式进行查找（只查找一层，工作量不大），很快就能够确定一个动作了。

如果你仍然觉得这些内容不够生动，那就来看一个形象但不够恰当的比喻吧。现在，我们手里有一个课题：研究孩子在某一时刻吃什么食物，能使他在小学阶段的各种评测中得到更高的分数。例如，在孩子即将入学的时候，他吃什么食物，会对他未来的发展产生影响？作为家长，如果想以科学的态度来解答这个问题，就要研究一下，入学的下一时刻会有哪些状态，以及在这些状态下孩子吃什么才好。因学校而异，在入学的下一时刻，有 30% 的可能是上文化课，有 50% 的可能是上体育课，有 20% 的可能是参加义务劳动。开个玩笑，就算只能吃馒头，也可以选择吃 1 到 10 个不等。

- 如果只吃 1 个馒头，很快就饿了。结果是，在这三种课程安排里，都只能得到 0 分。
- 如果吃 5 个馒头：上文化课，很快就饿了，只能得 60 分，不过比只吃 1 个馒头强；上体育课，能跑能跳，能得 90 分；参加义务劳动，力气十足，能得 90 分。这样一来，这一时刻的瞬时得分的数学期望值是 $60 \times 0.3 + 90 \times 0.5 + 90 \times 0.2 = 81$。
- 如果吃 10 个馒头：虽然很撑，但是上文化课坚持的时间长一些，学到的东西也多一些，能得 70 分；上体育课，可行性严重降低（此时如果做剧烈运动，容易导致阑尾炎或者其他疾病），只能得 20 分；参加义务劳动（不是剧烈运动），不仅可以完成，还比吃 5 个馒头持续

的时间长，得到 100 分。这样一来，这一时刻的瞬时得分的数学期望值就是 $70 \times 0.3 + 20 \times 0.5 + 100 \times 0.2 = 51$。

根据下一刻的课程安排概率（我们无法预测具体的课程安排），仍然可以得到吃下去的食物与得分的值，从而确定在某种饮食策略下，这一时刻、下一时刻、下下一时刻的得分的数学期望值，据此反推在入学的时候吃什么才好。

如果动态规划的方法不适用，该怎么办呢？在第 4 章中我们就来聊聊，对于不满足 MDP 的对象，应该用什么方法来做状态估值。

第 4 章　蒙特卡罗法

蒙特卡罗法（Monte Carlo Method）是一种可以帮助我们训练机器人行动策略的方法。一些资料中称之为“蒙特卡洛法”，也有一些资料中称之为“蒙特·卡罗法”。由于是音译，咱们就不深究了。也可以不去翻译这个词，直接叫它“MC 方法”——内行人都知道你在说什么。

蒙特卡罗法，也叫作统计模拟方法，是 20 世纪 40 年代中期由于科学技术的发展和电子计算机的发明而被提出的一种以概率统计理论为指导的非常重要的数值计算方法。蒙特卡罗法是指使用随机数（或者更常见的伪随机数）来解决很多计算问题的方法，与它对应的是确定性算法。蒙特卡罗法在金融工程学、宏观经济学、计算物理学（例如粒子输运计算、量子热力学计算、空气动力学计算）等领域有广泛的应用。

4.1　历史由来

蒙特卡罗法的名字让人有点不明所以，主要还是因为它的身世略显神秘。

早在 20 世纪 40 年代，为了研究原子弹制造过程中产生的中子扩散问题，美国的两位著名数学家冯·诺依曼[①]和乌拉姆[②]提出了一种在工程的角度实现模拟计算的方法，并以“蒙特卡罗法”为其命名。“蒙特卡罗”源于一座同名的摩纳哥公国小城，它位于地中海之滨，有“赌博之城”的称号。这个方法为什么被称为“蒙特卡罗”，传说有很多，其中之一是：因为这个项目是保密的，所以要找个代号来代替具体的方法名字，而非使用学术界惯用的命名方法；由于赌博过程中的很多行为可以用量化统计的思想来研究，其形式与蒙特卡罗法的本质相近，就使用了这个名字。我觉得，这个传说还是有一定可信度的。

实际上，蒙特卡罗法的思想并不复杂。它所秉承的观点是这样的：因为我们没有办法开启“上帝视角”，所以，我们不知道各种状态转化的概率是多少。那么，我们能不能通过反复实验进行统计，当统计的次数足够多的时候，就用这个统计的转化概率的近似值来代替模型中相应的真实值呢？其实是可以的——“实践出真知”有了另一种解释。

说起来，蒙特卡罗法不能算是冯·诺依曼和乌拉姆首创的，因为它是在长期数学实践应用中逐步总结出来的一种方法。历史更久远、更著名的例子是“蒲丰投针实验”。

---

① 约翰·冯·诺依曼（John von Neumann，1903 年 12 月—1957 年 2 月），美国著名的数学家、计算机科学家、物理学家，20 世纪最伟大的科学家之一，被后人称为“计算机之父”。曾参与研制世界上第一台电子计算机和第一枚原子弹。

② 斯塔尼斯拉夫·马尔钦·乌拉姆（Stanislaw Marein Ulam，1909 年 4 月—1984 年 5 月），美国著名的数学家。曾参与曼哈顿工程，研制原子弹。第二次世界大战后，又参与研制氢弹。

“蒲丰投针实验”是由法国数学家蒲丰[①]完成的一个有趣的数学实验。如果要计算圆周率，该怎么办？方法有很多。不过，我们能抛开复杂的测量过程、精确的量具、高深莫测的算法，用简单易行的方法得出圆周率吗？其实是可以的。

据说故事是这样的。1777 年的一天，蒲丰把一些朋友请到家里。他在一张很大的白纸上画了很多条等距的平行线，又拿出了许多质量均匀、长度恰好是平行线间距的一半的小针，请朋友们把针随意投到白纸上，他则在一旁观察投出的每一根针是否与平行线相交。当时的结果是这样的：共投针 2212 根，其中 704 根与平行线相交。于是，蒲丰计算了两数之商 $2212 \div 704 \approx 3.142$，然后宣布，这就是圆周率 π 的近似值。

其实，在做这个实验的时候，针的长度不一定是平行线间距的一半。可以扩展一下，把条件设置得宽泛一些，数值也未必要像故事里说的那样。取一张白纸，在上面画很多条间距为 d 的平行线。取一根长度为 l（$l \leqslant d$）的针，将这根针从这张纸的上方向纸上随机投掷 n 次，记录针与直线相交的次数 m，可以得到一个神奇的结果：

$$\lim_{n\to\infty} \frac{2l}{d} \cdot \frac{n}{m} = \pi.$$

也就是说，当投掷次数趋近于无穷的时候，这个比值无限趋近于圆周率 π。

据说在 1901 年，一个名叫拉泽里尼的人，在投掷 3408 次后，得到了 $\pi \approx 3.1415929$ 的小数点后 6 位的精确数字。在这个试验中，没有圆形出现，投掷的动作也是随机的，为什么能够把圆周率 π 的值测量出来呢？其实，在看似简单的现象背后，是一系列具体、扎实的数学逻辑证明，如图 4-1 所示。

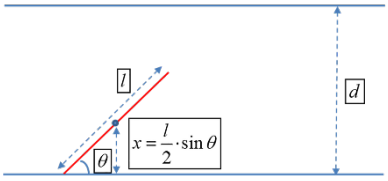


图 4-1　针与直线相交的情况

假设针与距离其最近的直线的夹角为 θ。取针的中点，并以该中点向距离其最近的这条直线作长度为 x 的垂线段，则有如下观测结果。

- 当 $x = \frac{l}{2} \cdot \sin\theta$ 时，就是图 4-1 中的样子，此时是相交和不相交的临界点。
- 当 $x \leqslant \frac{l}{2} \cdot \sin\theta$ 时，针和直线将会相交。因为针的位置是随机的，所以，x 在 $(0, \frac{d}{2})$ 上服从均匀分布[②]，θ 在 $(0, \frac{\pi}{2})$ 上也服从均匀分布。在这个时候，π 是未知数，是根据弧度的定义写出来的。

① 蒲丰（George-Louis Leclerc de Buffon，1707 年 9 月—1788 年 4 月），法国数学家、自然科学家。
② 均匀分布的意思是，在一个区间里，每个值出现的概率是一样的。

如图 4-2 所示，由 θ 和 x 两个维度，可以画出一个平面直角坐标系，θ 是横轴，x 是纵轴。根据 θ 和 x 的分布，就可以确定针是否与平行线相交。θ 的最大值是 π，x 的最大值是 $\frac{d}{2}$，这相当于实验中的全部事件的分布范围。当 $x \leqslant \frac{l}{2} \cdot \sin\theta$ 时，就是阴影部分所示的事件。

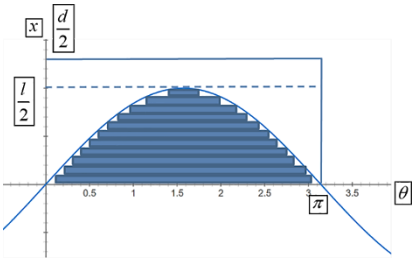


图 4-2　求解面积问题

可以通过求解这个阴影部分的面积来求出 $\frac{m}{n}$。这相当于，随机向这个矩形区域扔小球，坐标是 (θ, x)，取值概率是完全均等的，即小球落在这个区域的任何位置的概率是均等的。因此，如果扔小球 n 次，有 m 次小球落在如图 4-2 阴影部分所示的范围内，那么 $\frac{m}{n}$ 就是阴影部分的面积与矩形面积的比。

阴影部分面积与整个矩形面积的比 p，可以写作

$$p = \frac{\frac{1}{2}\int_0^{\pi} \sin\theta \, \mathrm{d}\theta}{\frac{d}{2}\pi} = \frac{2l}{\pi d} \approx \frac{m}{n},$$

所以，可以得到

$$\pi \approx \frac{2l}{d} \cdot \frac{n}{m}.$$

真是精妙绝伦。

这就是有关蒙特卡罗法思想的一个著名的实验——用多次实验逼近客观分布真实值的统计学应用。

4.2　状态估值

蒙特卡罗法是一个用于解决估值问题的统计学方法。

在第 3 章中，我们讨论了如何使用动态规划法来做状态估值和策略优化。这种方法的缺点在于，它是 Model-Based 的，对条件的要求比较苛刻。那么，对于不能提供 Model 的环境，该怎么办呢？蒙特卡罗法就派上用场了。例如，有一个机器人，它只知道在一个游戏环境中自己的样本空间和状态空间的取值范围，而不知道它们的客观分布，那么，它第一步应该做什么呢？最简单的思路是跟随动态规划的步伐，把这个 Model“试探”出来吗？这个思路在理论上是可行的。

赋予这个机器人一个策略 π（即使这个策略非常不靠谱都没关系），它在起始状态 s_1 下，一定能够走出一条路：

$$s_1 \rightarrow a_1 \rightarrow r_1 \rightarrow s_2 \rightarrow a_2 \rightarrow r_2 \rightarrow s_3 \rightarrow \ldots \rightarrow s_T.$$

最后一个状态 s_T，对一个 Episode（片段）来说，就是终止状态了（不管是通关了，还是“死翘翘”了）。

在这样一个序列产生以后，应该可以试着对某个状态 s 的价值进行评估：

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^{T-1} r_T.$$

对任何一个状态，我们都可以试着通过上面这个 G_t 的表达式来计算它的估值。

从形式上看，一个状态的估值就是根据当前的策略一步一步向后走，用每一步获得的奖励值乘以折扣系数 γ。走得越“远”，折扣系数就越大。也就是说，越“远”的奖励，与本次决策的关系越小。现在的问题在于，这条“路”每走一遍，就会得到一个序列，然而，这样偶然走出来的一条“路”，能客观地描述这个状态的价值吗？

我们知道，一个状态的价值需要通过类似加权平均的概念才能获得。在蒙特卡罗法中，要想计算一个估值，也应该采用类似的方法。没错，应该这样写：

$$v_\pi(s) = E_\pi[G_t | s_t = s].$$

这个表达式表示，对状态 s 的估值，需要在策略 π 的驱动下去求数学期望值。不严谨地说，如果用策略 π 玩这个游戏，每次都会产生一个与其他序列完全不同的序列

$$s_1 \rightarrow a_1 \rightarrow r_1 \rightarrow s_2 \rightarrow a_2 \rightarrow r_2 \rightarrow s_3 \rightarrow \ldots \rightarrow s_T.$$

对每个序列中的每个值进行价值评估，计算方法为

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^{T-1} r_T.$$

这样的事情能做多少次就做多少次，尽可能多地去做，就像 4.1 节中提到的投针实验一样。每个得到的状态，都有到达的次数和相应的价值估值，因此，对它们求加权平均值即可。

根据大数定律，这个加权平均值会向数学期望值逐步靠近。在一开始，偏差可能会比较大，距离它的真实值会非常远，但是，逐步靠近的趋势是确定的。

如果上面这个过程可以实现，那就是一个 MDP 问题，可以使用动态规划方法来解决了。不过，在这里没有必要这样做，因为如果要这样做，就要等待上面这个序列试探很多次后停下来、通过统计得到 MDP、再通过动态规划得到策略的过程完成——从计算效率的角度看，这不是一个经济的方法。

4.3　两种估值方法

为了保证估值计算的有效性，蒙特卡罗法会使用更适合自己的估值方法。蒙特卡罗法有两种估值方法，分别是 First-Visit Monte-Carlo Policy Evaluation（首次访问蒙特卡罗策略估值）和 Every-Visit Monte-Carlo Policy Evaluation（每次访问蒙特卡罗策略估值）。

4.3.1　首次访问蒙特卡罗策略估值

对任意状态 s 来说，如果在一个完整的 Episode 中第一次碰到它，就对计数器 $N(s)$ 进行加 1 的操作。当然，在一开始，所有状态的 $N(s)$ 的值肯定都是 0。随着访问次数的增加，每个状态的 $N(s)$ 的值将交替增加。在对 $N(s)$ 做加 1 操作的同时，需要对另一个用于记录总体奖励值的函数 $S(s)$ 进行加 G_t 的操作（G_t 的计算方式参见 4.2 节）。然后，可以得到估值

$$V(s) = \frac{S(s)}{N(s)}.$$

也就是说，只要在 Episode 中访问一个状态 s，且它在这个 Episode 中第一次出现，就进行“一揽子”操作：

$$\begin{aligned} N(s) &= N(s) + 1; \\ S(s) &= S(s) + G_t; \\ V(s) &= \frac{S(s)}{N(s)}. \end{aligned}$$

只要这么持续下去，根据大数定律，最终 $V(s)$ 肯定无限趋近于它的数学期望值

$$V_\pi(s) = E_\pi[G_t | s_t = s].$$

这个过程，和不断地扔一个匀质硬币，最终正面朝上和反面朝上的概率都无限接近 $\frac{1}{2}$ 的过程没有什么区别。

可以看出，这个方法非常简单，只有一个小小的问题：没办法在刚碰到 s 的时候就立刻进行操作，因为 G_t 一定是在这次 Episode 结束后才能得到的。也就是说，在执行一个 Episode 后，机器人需要回过头来，从第一个状态开始，逐个进行 $N(s)$、$S(s)$、$V(s)$ 的计算——这是首次访问蒙特卡罗策略估值法唯一看上去不够美的地方。

4.3.2　每次访问蒙特卡罗策略估值

每次访问蒙特卡罗策略估值，和首次访问蒙特卡罗策略估值非常相似。每次访问蒙特卡罗策略估值的评估方法，就像它的名字一样：第一次的访问策略就是针对第一次碰到这个状态的情况进行估算，后面再碰到，就不进行估算了；对每一次的访问策略，采用“一视同仁”的方式，即，第 1 次碰到状态 s_1 要做评估，第 2 次碰到状态 s_1 也要做评估，即便是第 100 次碰到状态 s_1，该

怎么评估，就怎么评估。

$$s_1 \to a_1 \to r_1 \to s_2 \to \ldots \to a_1 \to r_{120} \to s_1 \to \ldots \to s_T.$$

注意这个序列中的下标 1、2、*T*……它们表示的不是序数，而是不同的状态、不同的动作。因此，在这个序列中第二次出现的 s_1 表示机器人转了一圈，回到了前面出现过的一个状态。这个时候，同样按照

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^{T-1} r_T$$

的方式，分别对两次的 s_1 做估值计算即可。然后，依然按照

$$N(s) = N(s) + 1,$$
$$S(s) = S(s) + G_t,$$
$$V(s) = \frac{S(s)}{N(s)}$$

的原则，对状态 s_1 的估值进行更新计算。

不管是首次访问蒙特卡罗策略估值，还是每次访问蒙特卡罗策略估值，都会在训练过程中无限趋近于其真实估值。我在用蒙特卡罗法做游戏机器人训练的时候，为了编码方便，没有考虑是不是首次访问，都是碰到一次就进行一次估算。其实，首次访问蒙特卡罗策略估值的代码未必很复杂，这只是我的一个习惯。

4.3.3 增量平均

首次访问蒙特卡罗策略估值和每次访问蒙特卡罗策略估值，都可以通过在程序里设置计数器的方式解决求和及求平均值的问题。而在程序中，为了使得计算方便、表达简洁，通常会把这个更新公式写成它的增量平均（Incremental Mean）形式。

可以用这样一个公式进行更新：

$$V(S_t) = V(S_t) + \alpha\big(G_t - V(S_t)\big).$$

这个公式看上去要比本章前面介绍的公式简单得多，其中没有次数计数器 $N(s)$，也没有累加的 $S(s) = S(s) + G_t$。那么，这个公式表达的意思和本章前面介绍的公式一样吗？是一样的。下面我们一起分析一下。

我们先来看一个更具有广泛性的表达式，看完你就会知道这个陌生的 α 是怎么冒出来的了。例如：有一个序列 x_i，从 x_1 到 x_k，一共有 k 个数值；还有一个序列 μ_k，用于表示任何一个从 x_1 到 x_k 的平均值。这个表达式应该怎么写呢？

$$\mu_k = \frac{1}{k}\sum_{j=1}^{k} x_j.$$

是的，就是一个循环求和运算及一个平均值运算。这个表达式的形式和 4.3.2 节中的 $N(s)$、$S(s)$ 和 $V(s)$ 的形式如出一辙，$N(s)$ 就是这个表达式中的 k，$S(s)$ 就是这个表达式中的 $\sum_{j=1}^{k} x_j$，$V(s)$ 就是这个表达式中的 μ_k。

对这个表达式稍做修改，

$$\begin{aligned}\mu_k &= \frac{1}{k}\sum_{j=1}^{k} x_j \\ &= \frac{1}{k}(x_k + \sum_{k=1}^{k-1} x_j) \\ &= \frac{1}{k}(x_k + (k-1)\mu_{k-1}) \\ &= \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1}),\end{aligned}$$

即

$$\mu_k = \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1}).$$

现在，回过头来看下面这个表达式，是不是明白多了？

$$V(S_t) = V(S_t) + \alpha\big(G_t - V(S_t)\big).$$

x_k 不就是每次收集到的新的 Episode 的估值 G_t 吗？等号后面的 $V(S_t)$ 不就是旧的 $V(S_t)$ 的值吗？等号前面的 $V(S_t)$ 就是新的 $V(S_t)$ 的值。所以，α 就是 $\frac{1}{k}$，也就是计数器的值的倒数。例如，$\alpha = 0.0001$ 表示 $V(S_t)$ 是 10000 个值的平均值。

4.4　弊端

蒙特卡罗法的过程，是一个通过不断采样让样本丰富起来，进而比较准确地估算出由策略 π 驱使的某个状态的估值 $v_\pi(s)$ 的过程。一旦准确地计算出这个值估，就几乎可以沿用动态规划的方法进行策略优化了。简单地说，通过多次实验进行统计，计算状态估值，一旦状态估值 $v(s)$ 能够估算准确，就能评价一个动作的估值 $q(s,a)$。只要这一点能够达成，那么，一个好的策略就会变成“在某状态下，查看 $q(s,a)$ 中的哪个 a 在当前给定的状态 s 下估值最高”的问题了，原因在于，只要 $q(s,a)$ 是准确的，就表示其中包含远期收益（而不是瞬时收益），$q(s,a)$ 是一个可以信任的估值，而这个过程，就是由蒙特卡罗法“试”出来的。

蒙特卡罗法的优势在于，它不是以模型为基础的，而是通过多次试探求平均值的方法让估值逐步变得准确的。于是，问题产生了。在统计中，尽管随着实验次数的增加，平均值会逐渐稳定在一个值附近（例如，在扔硬币时，一定会从第一次正面朝上或反面朝上的概率为 100% 开始，随后统计结果大幅波动，最终逐步无限逼近正面朝上和反面朝上的概率均为 50% 这个统计平均值），

然而，在这个过程中，前几次实验结果的方差一定大得惊人。蒙特卡罗法也没办法避免这个问题，在开始的几十个甚至上百个 Episode 中，这种方差会像梦魇一样伴随着我们。这些最初产生的状态估值都严重偏离真实的估值，是不可信的。

不是说蒙特卡罗法不好，而是任何方法都会在一些场景中体现出局限性。这一点是需要我们注意的。

4.5 本章小结

本章介绍的是蒙特卡罗法的估值方法与优化方法。蒙特卡罗法有很多可取之处，毕竟比起动态规划法，它不要求一个完整的 Model（或者环境满足 MDP），也就是说，它对环境的适应性更强。每一次 Episode 结束，蒙特卡罗法都会对每个状态的值进行估算。随着试探次数的增加，这个估值将越来越客观、准确。

还是用 3.3 节中那个吃馒头的例子来打比方。作为一名具有科学头脑的家长，现在我们有了新的武器——虽然不知道学校里会安排什么课程，但是没关系，我们可以查资料。在网上搜索与“一名即将入学的孩子的饮食对其成绩的影响”有关的内容，能得到什么样的结果呢？

假如，我们找到 100 篇文章，其中分别有 100 个不同的且真实的故事。

- 有 10 篇文章中说，“孩子入学时吃 10 个馒头，每门功课都得 95 分，保送重点初中”。这就厉害了。
- 有 70 篇文章中说，“孩子入学时吃 10 个馒头，结果得了盲肠炎，直接住院休学 6 年”。这下麻烦大了，因为从数值上看，得分收益为 0。
- 有 20 篇文章中说，“某著名权威专家研究表明，孩子入学时吃 10 个馒头，精力旺盛，每门功课都得 85 分，且没有不良影响”。

这时，作为家长，看看自己的孩子，再看看手里装着每个 2 斤重的 10 个山东饸面大馒头的袋子……把这些馒头塞到孩子嘴里，6 年后，孩子的学习成绩会怎么样？坦白地说，这对孩子在小学阶段的这个不可复制的 Episode 来说，没什么意义——孩子有可能每门功课都得 95 分，也有可能每门功课都得 85 分，还有可能休学了（每门功课都得 0 分）。但是，这组数据仍然具有统计学意义：在每 100 个孩子中，如果每个孩子都在入学时吃 10 个馒头，有 10% 的孩子每门功课得 95 分，有 70% 的孩子每门功课得 0 分，还有 20% 的孩子每门功课得 85 分，对这组数据计算算数平均值……是的，只有 26.5 分（$95 \times 0.1 + 0 \times 0.7 + 85 \times 0.2 = 26.5$）。暴饮暴食易伤脾胃啊，还是查查那些吃 9 个馒头、8 个馒头的孩子的成绩如何，再决定自己的孩子在入学时吃几个馒头吧——开个玩笑。

蒙特卡罗法的缺陷也是很明显的，那就是必须等待一个 Episode 结束才能开始进行估算（这样的估算效率不是很高）。能不能更快、更直接地得到这些状态的估值，从而高频度地进行策略优化呢？这是一个新的问题，答案也是有的，就是第 5 章中将要介绍的时间差分。

第 5 章　时间差分

动态规划法的思路很简洁，但实际操作起来有一定的难度，需要通过模型在一棵树上从下到上传递，才能把状态的估值计算准确。蒙特卡罗法的操作很简单，只不过需要在一个 Episode 结束后才能返回一个状态的估值更新。那么，能不能用一些更直接的方法返回估值呢？也可以，那就是时间差分（Time Difference，TD）法。

时间差分法的思路与蒙特卡罗法不同，它不希望机器人通过一个完整的 Episode 试探之后才对一个状态进行估算，它可以走一步就估算一次。也许读者朋友会有点纳闷：走一步就估算一次，岂不是连这条路能不能走到底都不知道？这估值准吗？

不得不承认，这种方法确实有让估值不准确的嫌疑。但是，通过蒙特卡罗法得到的估值，不仅不准确，而且方差特别大。所以，这至少不应该被称为时间差分法的劣势，我们不需要对时间差分法太过苛责。况且，通过不断尝试，根据统计学中的大数定律，虽然一个状态的估值每次都会不同，但终究会令其平均值趋近于某个数字，于是，时间差分法就能够逐渐把这个值估计算准确了。

需要强调的是，时间差分法的优势相对更为明显，因为它的学习效率看起来比动态规划法和蒙特卡罗法高一些——做一个动作，很快就能让机器人更新“大脑”里的信息。那么，时间差分法中有哪些可用的算法呢？

在本章中，我们就将介绍两个最为经典的算法，一个是 SARSA 算法，另一个是 Q-Learning 算法。它们从诞生到现在，一直都是强化学习教程中不可或缺的内容。

5.1　SARSA 算法

“SARSA”这个名字看上去很有趣，其中的字母都是大写的。不用我多说，大家也能猜个差不离：它不是一个英文单词，而是一个词组或者多个单词的缩写。确实，从结构上看，SARSA 是

$$S \to A \to R \to S \to A$$

的缩写。一言以蔽之，在 SARSA 算法的眼中（或者说计算逻辑中），在状态 S 下，要想把动作 A 的估值 $Q(S, A)$ 计算准确，需要研究

$$S \to A \to R \to S' \to A'$$

这个序列，并对这个序列中的估值进行调整，从而达到使贝尔曼方程收敛的目的。

更新方程是这样的：

$$Q(S,A) \leftarrow Q(S,A) + \alpha\big(R + \gamma Q(S',A') - Q(S,A)\big).$$

用“白话”解释，意思是：SARSA 算法的表现形式是 $Q(S,A)$，即对某个状态下的一个动作做估值计算。在状态 S 下，做一个动作 A，它的估值 $Q(S,A)$ 需要的迭代计算逻辑是当前的 $Q(S,A)$ 与一个 α 倍的叠加值相加。对这个 α，我们应该不陌生，在第 4 章介绍蒙特卡罗法时，我们刚刚见过它——α 的值越小，说明“记忆”的周期越长，也可以说，对于估值，“平均”的周期越长。这个叠加值，是用瞬时回报 R 和 γ 倍的下一个状态下的动作估值 $Q(S',A')$ 相减得到的。

γ 是折扣系数，这一点在介绍动态规划法、蒙特卡罗法及 MDP 时都提到过。它表示对于长期回报的好恶。γ 的值在 0 和 1 之间：值越大，说明算法越“远视”，即对远期的回报越重视，越倾向于把它们的得失算在当前决策的头上；值越小，说明算法越“无视”长期回报；如果是 0，相当于只看“现世报”（我们通常觉得这样做不太好）。

我们讨论一下 $\alpha = 0$ 时的情况。这时，从表达式上看，就是小朋友之间互相许诺“一百年不许变”情况——后半部分肯定是 0。同时也说明，k 趋近于无穷。如果 k 的值非常大，那么在将每次学到的新的估值合入后计算平均值时，对平均值的影响是微乎其微的（因为 k 在计算平均值的过程中是作为分母的）。

如果 $\alpha = 1$，相当于每次 $Q(S,A)$ 都只关注 $R + \gamma Q(S',A')$ 的值，且只将这个值作为当前的估值。这时，原来学到的东西全都没有记住——也就是 $k = 1$ 的情况。

两者相较，哪个更好呢？还真不能一概而论。这似乎只取决于我们想通过什么样的时间片来实现平滑。α 的值越小，则估值变化越慢、越平滑，也就越能参考更长周期的变化平均值；α 的值越大，则变化越快、越剧烈，也就越关注最近几个时间周期的变化平均值。这一点和蒙特卡罗法没有太大的区别，只不过，时间差分法走一步（一个 Step 循环）就进行一次更新，不必等到整个 Episode 结束再进行更新。我倾向于将 α 取得小一些，让变化的平滑度高一些。

5.1.1 SARSA 算法的伪代码

SARSA 算法的逻辑并不复杂，下面我们一起来看看 SARSA 算法的伪代码。

```
初始化函数 Q(s,a)，令任何一个终止状态的 Q 值都为 0
对于每个 Episode 循环
    设置初始状态 S
    根据 Q 函数与状态 S，选择动作A（例如 ε − greedy 方法）
    对于每个 Step 循环
        做一个动作A，观测 R 和 S'
        根据 Q 函数和状态 S'，选择动作A'（例如 ε − greedy 方法）
        赋值：Q(S,A) ← Q(S,A) + α[R + γQ(S',A') − Q(S,A)]
        赋值：S ← S', A ← A'
    直到 S 是终止状态
```

这段伪代码非常短，只有 10 行，内容也很清晰，因此，在这里只对其进行简单的解释。

首先，函数 $Q(s,a)$ 的初始化其实是一个随机性比较强的过程，其原因还是在于冷启动时没有参考数据。因此，不妨设置一个空列表，这样在查表的时候会得到任意一个状态的任意动作的估值 $Q(s,a)=0$。这个问题在实际应用中影响不大，反正最开始估值的准确性不是那么重要。终止状态必须是 $Q(s,a)=0$。

顺便说一下，这里的“冷启动”不是指直接去按计算机的电源键，它们只是同名而已。在这里，冷启动（Cold Boot）是一个技术概念，指的是系统的一个状态。不管是我们在购物网站看到的推荐系统，还是我们现在正在学习的强化学习，都是需要进行智能输出的系统。只不过，推荐系统要输出的是更符合用户需求的、更贴心的产品列表，而强化学习系统要输出的可能是机器人的一个智能的动作。但不管怎么样，在它们诞生时，在它们从客观世界中学习和认知事物之前，它们很难有能力输出高质量的列表和动作。我们知道，机器人需要通过大量的 Episode 进行估值计算，才能准确地计算状态的估值；同样，推荐系统要反复为用户推荐不同的商品，并从用户的反馈信息中逐步总结用户的好恶。这是一个学习的过程。我喜欢打趣说，二者在脑袋空空地去认识这个世界的时候，都会或多或少表现出“人工智障”的气质。这里所说的冷启动，就是指这种还没有学到任何东西的状态。机器学习系统冷启动是一个很有趣的研究方向。有不少学者正在研究：能否让机器学习系统在启动时尽快获得一些基本的知识和判断能力，从而在一开始就表现得好一些，而不是“从零开始”学习。如果对这个话题感兴趣，可以找一些相关的资料来阅读。下面，我们言归正传。

接下来，让机器人在环境中不停地“玩游戏”（如图 5-1 所示）——从“诞生”时就开始玩，直到出现 Terminal 的状态，也就是伪代码中所述的终止状态。这就是一个完整的 Episode 过程。

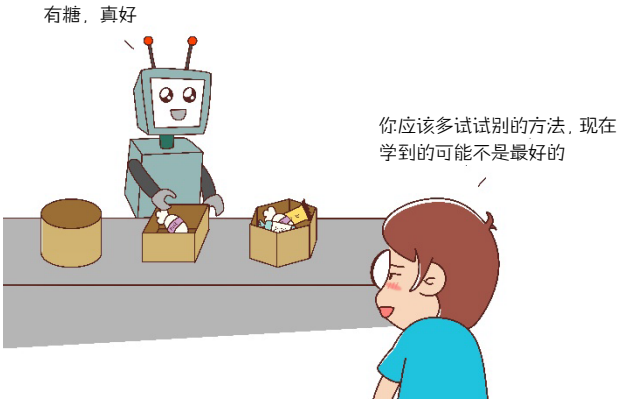


图 5-1　爱学习的机器人

对每个 Episode 来说，都要初始化一个状态 s，让机器人开始学习（这可以通过 $Q(s,a)$ 和 ε–greedy 方法实现）。ε 是一个希腊字母，读作“爱普希龙”。ε–greedy 方法的要点如下。

- 令 ε 取一个 0 和 1 之间的数字——看这架势就觉得和概率有关系。没错，就是概率。

- 以 $1-\varepsilon$ 的概率，让机器人做 $a^* = \underset{a \in A}{\text{argmax}} Q(s,a)$ 的动作，也就是 $Q(s,a)$ 中能够产生最高估值的动作 a。
- 以 ε 的概率进行试探，随机做一个动作，以兼顾试探和最优动作选择的方式不断提高机器人输出动作的质量。

在以上三个要点中，第二个和第三个涉及同一个步骤，只不过，在一次行动中只能二选一。

ε 的值取多少才合适呢？不能“一刀切”。不过，我们可以确定的是，ε 的值越大，试探的意味就越强，ε 的值越小，动作就越“保守”、越相信自己学到的知识。因此，我认为，从定性的角度看，ε 以变化的形式出现比较好。

假如我们要训练一个机器人，让它玩一款游戏。可以考虑在前 1000 个 Episode 中使 $\varepsilon = 1$，让机器人充分收集信息；在 1001 到 2000 个 Episode 中使 $\varepsilon = 0.8$，让机器人逐步利用已经学到的经验来输出质量比较高的动作。接下来，逐步减小 ε 的值，例如在 10000 个 Episode 之后使 $\varepsilon = 0.001$。这时，机器人大概率相信自己已经学到的知识，而这 0.1% 的概率，表示寄希望于机器人不管有多少见识，还是能够突发奇想，做一些“放卫星”的事情。

在每个 Step 中，机器人智能体都输出一个动作 A，同时观测 R 和 S'，然后使用同一策略，在 S' 状态下做动作 A'，用

$$Q(S,A) \leftarrow Q(S,A) + \alpha(R + \gamma Q(S',A') - Q(S,A))$$

去更新 $Q(S,A)$。需要注意的是，“$\leftarrow$”后面的部分就是用类似查表的方式，直接把基于当前策略的 $Q(S',A')$ 估值查出来并进行叠加，而不是在这一刻观测 R' 的值。

SARSA 算法是一种 On-Policy（在线）算法，也是基于一种已有策略的估算，如图 5-2 所示。因此，在进化的过程中，机器人能知道 $S \to A \to R \to S' \to A'$ 这个确定的序列的样子，并能根据它把估值计算准确。

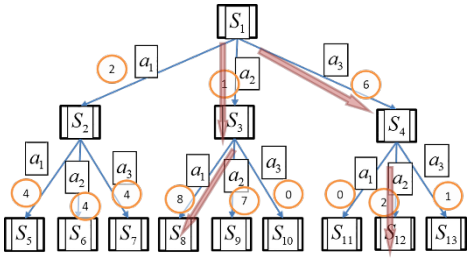


图 5-2　SARSA 算法

在图 5-2 中，S_1 到 S_{13} 只表示不同的状态，不表示严格意义上的次序。如果想要估算在 S_1 状态下做动作 a_1、a_2、a_3 的估值，除了要把过去学到的 $Q(S,A)$ 保留下来，还要参考根据当前策略所到达的状态及在这个状态下动作的估值 $Q(S',A')$，也就是

$$Q(S,A) \leftarrow Q(S,A) + \alpha(R + \gamma Q(S',A') - Q(S,A))$$

的含义。至于 $Q(S',A')$ 的估值，仍旧通过在一次一次的 Episode 中学习得到。例如，要想估算 $Q(S_1,a_2)$ 的值，就要用 $Q(S_1,a_2)$ 原来的估值大小和 $\alpha(R+\gamma Q(S',A')-Q(S,A))$ 做求和运算。在当前策略下，$Q(S_1,a_2)$ 会迁移到 S_3 状态，并且在 S_3 状态下，a_1 是即将采纳的动作。假设超参数 $\alpha=0.01$，根据图 5-2 中 $R=1$ 的条件，$Q(S_1,a_2)$ 的更新就变成了

$$Q(S_1,A_2)\leftarrow Q(S_1,A_2)+0.01\times\big(1+\gamma Q(S_3,A_1)-Q(S_1,A_2)\big).$$

简言之，SARSA 评估的内容是在一个策略下的 $Q(S,A)$，方法是：评估当前状态的 $Q(S,A)$，要参考当前策略下的下一个状态的 $Q(S',A')$，通过这种将 Q 的值（奖励值）层层回溯的方式，逐步调整状态 $Q(S,A)$ 的估值。这就是时间差分法中比较典型的学习算法，实现方法非常简单。

5.1.2　SARSA 算法的优点和缺点

SARSA 算法的逻辑是非常简单的，与动态规划法和蒙特卡罗法相比有比较明显的优势。

- SARSA 算法不要求环境必须是 MDP 或者 Model-Based 的，也不像动态规划法那样基于 $P^a_{ss'}$ 来计算估值。SARSA 算法的估值是靠它自己一次一次尝试得出的。
- SARSA 算法的估值，反馈速度比较快，可以走一步就进行一次反馈，而不像蒙特卡罗法那样，必须等一个完整的 Episode 结束才能给出反馈。
- SARSA 算法的“视野”比蒙特卡罗法“窄”一些。因为蒙特卡罗法要“一条道走到黑”，所以其估值的方差要小一些，在变化上要平滑一些。

至于 SARSA 算法有没有缺点，我也说不准。不过，在实验中，尤其在简单的场景中，SARSA 算法的收敛速度不如 5.2 节将要介绍的 Q-Learning 算法快。

5.2　Q−Learning 算法

可以说，Q-Learning 算法是强化学习中最为有效、最具普适性的算法。尽管从名字上看不出它是如何工作的，但它仍然是一个 TD 算法，并且是 Off-Policy（离线）的——这一点与 SARSA 不同，SARSA 是 On-Policy 的。

先看公式，其形式和 SARSA 非常像，我们对比一下。

- Q-Learning 算法：$Q(S,A)\leftarrow Q(S,A)+\alpha(R+\gamma\max\limits_{a'}Q(S',A')-Q(S,A))$。
- SARSA 算法：$Q(S,A)\leftarrow Q(S,A)+\alpha(R+\gamma Q(S',A')-Q(S,A))$。

从形式上看，只有后半部分不同。SARSA 算法使用的是一个带有折扣的估值 $\gamma Q(S',A')$，而 Q-Learning 算法使用的是一个带有折扣的 $\gamma\max\limits_{a'}Q(S',A')$，差别就在“max”上。

Q-Learning 算法对 $Q(S,A)$ 的估值，同样参考了其后所能转移到的所有状态 S' 可能做的所有动作 a'，并认为其中的最大估值才是当前 $Q(S,A)$ 的估值。这个思路是非常明确的：当前状态 $Q(S,A)$ 的大小，取决于在它转移到的状态下，哪个 a' 能够取得最大的估值（奖励值）及估值为

多少。用一句话概括：既然曾经有人在这么做之后能够达到某个估值，那么策略就应该将这个动作作为输出，以谋求“利益最大化”。可以说，这是一种看上去比较“大胆”“贪婪”“冒险”的进化策略。

5.2.1 Q-Learning 算法的伪代码

我们来看看 Q-Learning 算法的伪代码。

初始化函数 $Q(s,a)$，令任何一个终止状态的 Q 值都为 0
对于每个 Episode 循环
    设置初始状态 S
    对于每个 Step 循环
        根据 Q 函数与状态 S，选择动作 A（例如 $\varepsilon-$greedy 方法）
        做一个动作 A，观测 R 和 S'
        赋值：$Q(S,A) \leftarrow Q(S,A)+\alpha\left[R+\gamma\max_{a} Q(S',a)-Q(S,A)\right]$
        赋值：$S \leftarrow S'$
    直到 S 是终止状态

这段伪代码和 SARSA 算法的伪代码非常像，主要的差别在对估值函数进行更新的表达式上。相信有 SARSA 算法做铺垫，对 Q-Learning 算法的功能，我们能通过伪代码了解得差不多（如图 5-3 所示）。

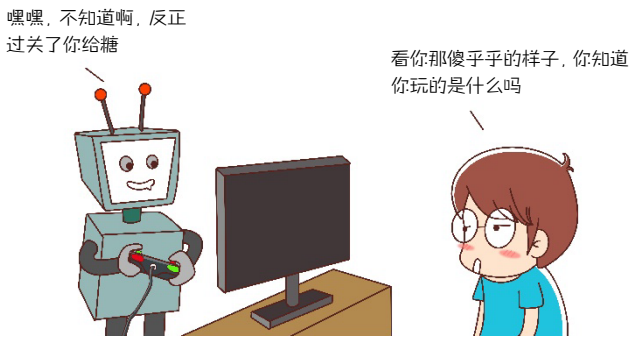


图 5-3 Q-Learning 算法

在 $Q(S,A)$ 的估值中，除了当前的直接奖励值 R，还有 $\max_{a'} Q(S',a')$。在这里没有强调 S' 和 a 是在同一个策略下产生的，相反，我们其实更希望这是不同策略下的 S' 和 a 的行为估值。因为只有这样，才会有比较丰富的 $Q(S',a)$，在其中挑选“max”的时候，才能够发现比较好的 S' 和 a 的值，从而把其时间序列前方的 $Q(S,A)$ 估算得更“准确”，或者说，更容易发现比较好的策略行为路径，进而得到比较好的策略。

还是用本书前面多次提到的那个吃馒头的例子来做类比。在每个决定要吃东西的时间点，都要查询一下，看看“吃几个馒头好”。即使查询结果是“某位同学上课前吃了 10 个馒头，在当堂

课上学习效果一般，课堂测验成绩为 30 分，但是由于馒头后劲较足，在第二堂课的测验中得到了 90 分的高分”，也不能排除“第二堂课还是糊里糊涂，最后得了 20 分、30 分”的情况，因此，还需要查询上课前吃 9 个、8 个、7 个等数量的馒头的情况，并关注这些孩子在第二堂课的测验中的得分。对查询结果进行综合比较，发现有一条记录是“吃了 7 个馒头，在第二堂课的测验中得了 100 分”，那么，就将 Q(上课前, 吃 7 个馒头) 的估值设为 100，然后勇敢地让孩子吃下 7 个馒头，高高兴兴上课去。

5.2.2　Q–Learning 算法的优点和缺点

Q-Learning 算法在强化学习中的应用场景很多，效果也都不错。近年来，有许多深度学习与 Q-Learning 算法结合的实例出现，它们都是在相同的理论基础上建立起来的。

Q-Learning 算法的优点如下。

- 因为使用“max”进行估值计算，所以，容易在比较简单的环境中发现最优路径、找到最好的策略。
- 实现简单，只需要对 $S \to A \to R \to S'$ 的值进行记录，并逐步进行更新。同时，其逻辑比较容易理解。

Q-Learning 算法的缺点如下。

- 几乎无法（或者很难）实现输出动作为连续值的场景。不仅 Q-Learning 算法存在这样的缺陷，SARSA 算法也存在这样的缺陷。其主要原因是，在更新公式

$$Q(S,A) \leftarrow Q(S,A) + \alpha(R + \gamma \max_{a'} Q(S',a) - Q(S,A))$$

  中，很难通过回溯的方式准确计算 $Q(S,A)$ 的估值。
- Q-Learning 算法中的“max”，原本的逻辑是：一个状态下的动作估值，完全取决于其下一个状态下的动作中估值最大的那个。但是，这不仅和原始定义

$$E[R(\pi(s|\theta)|s)] = \frac{1}{N}\sum_{i=1}^{N} R(a|s_i)\,\pi(a|s_i)P(s_i)$$

  是矛盾的，也和我们在日常生活中形成的认知是矛盾的。

对一个状态，要想估算它的价值，尚且需要使用数学期望值这样的概念，通过多次测量取平均值，因此，单纯以最大值来做评价是有失偏颇的。

举一个现实生活中的例子。假设你的手里有 1 万元，那么在这个时候，拿这笔钱去做什么比较好？这是一个投资策略问题——可以存到银行里，可以去旅行，可以去进修、学习，当然，也可以去买彩票。就说学习和买彩票这两件事情吧。在拥有 1 万元资金的情况下，如何评价“买彩票”和“进修学习”？单纯从

$$Q(S,A) \leftarrow Q(S,A) + \alpha(R + \gamma \max_{a'} Q(S',a) - Q(S,A))$$

的角度看，“买彩票”这个动作是有机会得到 100 万元的（确实有人中过 100 万元）。不过，直接用 100 万元来估计“买彩票”这个行为并把它当作估值，肯定是不合适的。这种情况被称作“过估计”（Overestimating），如图 5-4 所示。

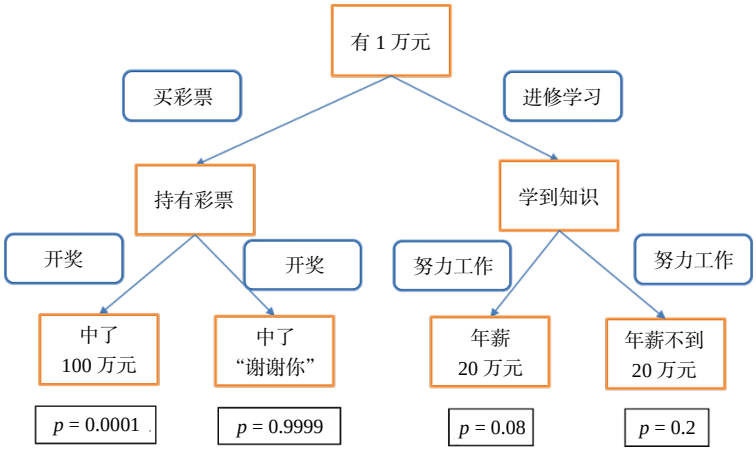


图 5-4　Q-Learning 算法的过估计

毫无疑问，其中有一个概率的问题。“中了 100 万元”是小概率事件，我们不应该将小概率事件的值作为一个动作的估值，因为这对每个人都不公平。当你手里有 1 万元的时候，比较靠谱的方式应该是投资自己，让自己学到更多的知识，提高认知水平或者业务能力。只有这样，你才能在未来大概率发挥自己的能力，创造更多的价值。在这个例子里，我只是示意性地提出年薪 20 万元的概率为 80%、年薪不到 20 万元（例如 0，即失业）的概率为 20%。即便如此，“进修学习”的数学期望值也有 16 万元。与之相对的是，用 1 万元“买彩票”的数学期望值只有 100 元——高下立见。

在这里举这个例子，主要就是为了说明，TD 算法中的 Q-Learning 会出现机器人对价值的“误解”问题。这是 TD 算法和 DP 算法的一个比较大的差异，也是 TD 算法与生俱来的缺陷，请大家务必注意。不过，也没必要谈虎色变，觉得 Q-Learning 是一种非常不靠谱的算法，它只是有暂时性的估值过高的问题而已。一方面，这个问题在目前的 Q-Learning 算法中不是永久存在的；另一方面，在这个问题被发现之后，深度强化学习的 DQN 算法族也采用了很多方法来应对。DQN 算法族针对这个问题提供的解决方案，将在后面讲解深度强化学习的部分介绍。在本节中，我们只讨论一下为什么在 Q-Learning 算法中这个问题是暂时性的。

我们继续看如图 5-4 所示的“买彩票”的例子。生活没有终点，“中了 100 万元”当然不会是生活的终点，因此，“买彩票”的价值可能暂时被高估了（价值 100 万元）。然而，在“中了 100 万元”之后，每个人的行为都是不同的——所谓“Off-Policy”，就是什么样的策略都有。例如，有的人拿这笔钱去投资，赚了 1000 万元，而有的人不仅把这笔钱挥霍一空，还欠了一屁股债。那么，

对“中了 100 万元”这个动作进行估算，会得到什么呢？是有人通过投资赚了 1000 万元，因此“买彩票”这个状态也被进一步高估至 1000 万元吗？

可能你要问了：前面就说这个状态已经是过估计的了，这样一来，岂不是更“过”了？如图 5-5 所示，就是要说明一个观点：不管这个状态的值是不是过估计的，如果在它的下面有新的状态出现，那么估计仍会继续，并会提供一个客观、公正的值来取代这个临时的值。而在这个新的状态下面，可能还有别的状态，这些状态中的最大值，仍然会在回溯中不断通过贝尔曼方程向上传递，影响前导状态的估值。

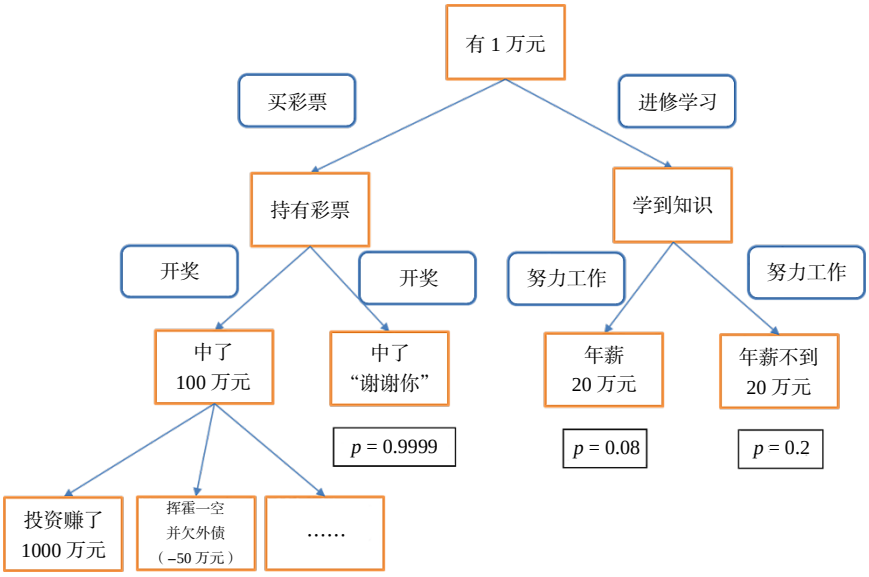


图 5-5　“治愈”Q-Learning 算法的过估计

当然，大概率出现的状态还是会大概率影响回溯进度的，因为这个状态会被大概率地“走”到，进而进行大概率的 max 采样。同时，α 也会影响 $Q(s,a)$，把 α 的值取得小一些，$Q(s,a)$ 对这些在小概率下取到的值就不那么敏感了（因为它会尽量保留先学到的值）。这种平稳的特性在深度强化学习中尤为重要，在本书后面的章节中会具体介绍。

Q-Learning 算法就介绍到这里，示例代码参见本书的 GitHub 页面。

5.3　On-Policy 和 Off-Policy

On-Policy 和 Off-Policy 是本章中不得不讲的两个概念。老实说，我真的不想突然扔出两个名词——这会让读者直接眼冒金星的。为什么时间差分法会与 SARSA 和 Q-Learning 这两个看起来相似但实际上并不相同的算法有关？尽管在本章前面的内容中提到过，在这里还是要做一些详细的解释。

所谓“On-Policy”，在强化学习的范畴内是指，在训练中用来计算估值并训练的策略和它所采用的所有 Transition 的策略是同一套策略。Off-Policy 与 On-Policy 相反，在训练中用来计算估值并训练的策略和它所采用的所有 Transition 的策略都不相同。

从这个角度看，SARSA 算法是典型的 On-Policy 算法，Q-Learning 算法是典型的 Off-Policy 算法。如果你非要问 On-Policy 和 Off-Policy 中更好的是哪一个，我认为，没办法直接作答。如果其中有一个非常明确地比另一个好，那么另一个就不应该存在；既然两者都存在，那么，原因一定是两者在不同的场景中有不同的优点和缺点、彼此之间不能完全互相取代。这样的思路才是辩证的。

例如，On-Policy 的 SARSA 算法，使用同一套策略在环境中做动作，并把 Transition 信息收集起来，每“走”一步，都要更新 $Q(s,a)$ 的估值。这个逻辑是非常单纯的，但缺点也很明显——“一根筋”的逻辑，使得 Transition 的前后相关性非常强，也就是说，一个由中途的某个 Step 产生的不太好的选择，会对未来若干步都产生比较深远的影响。打个比方：一个棋艺不高的人，一边自己和自己下棋，一边总结得失，那么，按照这种棋谱来学习的人，能有多大进步？其实，这个例子不算恰当，因为这个人只能看见自己是怎么落子的，却看不见别人是怎么落子的……所以，SARSA 算法无论如何都要使用 ε–greedy 策略来增加试探次数，不然，用空空的大脑“一根筋”走下去，真是危险得很。

Off-Policy 的 Q-Learning 算法，从态度上看，有点“贪婪”的意思，因为它每次都选择估值最大的动作进行更新，而不在乎策略是什么，因此，影响它的主要是每个状态的估值。按理说，Off-Policy 的数据来源是使用不同的策略试探得到的 Transition，所以比较丰富。然而，在 Q-Learning 算法里，这种情况并不明显（但是，当第 8 章中介绍的 DQN 的模型算法里有了 Replay Memory 之后，感觉就不一样了）。

5.4 On-Line 学习和 Off-Line 学习

在一些资料中会提到，强化学习有两种学习方式，分别是 On-Line 学习和 Off-Line 学习，或称在线学习和离线学习。所谓在线学习可以理解为，一个机器人程序能够一边玩游戏，一边在这个过程中收集样本并不断学习新的样本所提供的内容——“在战争中学会战争”。离线学习的方式与此相反，机器人程序需要先进行类似监督学习的学习，通过训练得到完整的策略，再用这个策略到游戏环境中去做动作。

感觉上，On-Line 学习的效率好像高一些，而 Off-Line 好像比较“笨”，不能一边玩游戏一边学习。但我觉得，不能直接这样理解。

做过服务器开发或者系统优化的读者朋友，可能会对效率的问题比较敏感，因为所有的 Batch（批）操作，无论是在 I/O 资源方面，还是在 CPU 资源方面，都更有优势。如果有一个无穷大的或者非常丰富的 Transition 样本库，Off-Line 的训练速度还是非常快的——相当于在一个海量存储环

境中大批量地做贝尔曼方程的更新，并且只做这件事。这种“训练”的效率当然很高，因为根本不用再让机器人去和环境做任何交互，只要通过不断更新把 $Q(s,a)$ 的值逐步估算准确即可。但是，这样做的前提是我们拥有大量的 Transition 信息。

现实是很残酷的。实际上，在绝大多数场景中，我们根本没有机会存储足够多的 Transition 样本。一个 Q-Learning 算法，即使在升级完成、变成深度学习的 DQN 以后，也只能保存最近的百十万个 Transition 信息。要想获得更多新的样本信息，只能一边玩游戏，一边忘记以前的，一边存储最新的，一边更新贝尔曼方程——这才是问题所在。

因此，绝大部分基于时间差分法的强化学习训练，都倾向于使用 On-Line 学习。On-Line 学习还有一个好处，就是在训练的过程中可以方便、清晰地看到实验结果。例如，一个机器人程序经过 3 小时训练，是已经变得比较聪明了，还是没什么进步？只有在环境中经过多个 Episode 的测试，才能有量化的结论。我们也希望机器人程序能够在实践中不断进步，也就是一边“锻炼”一边进步——这样学习效率显得比较高。如果完全依据已经生成的高质量的 Transition 信息来训练，那就属于监督学习或者模仿学习的范畴了，而不属于我们正在讨论的基于时间差分法的强化学习的范畴。

5.5　比较与讨论

在这一章中，我们重点讨论了强化学习领域的两个经典算法，一个是 SARSA 算法，另一个是 Q-Learning 算法，前者是 On-Policy 算法，后者是 Off-Policy 算法。由于 Off-Policy 本身的合理性，Q-Learning 算法的使用范围更广。

SARSA 算法和 Q-Learning 算法，都是典型的基于 TD 的算法，也叫作抽样回溯（Sampled Backup），而这会导致 Q-Learning 算法对动作产生过估计的问题。

1. Q-Learning 算法的思维方式

不可否认，Q-Learning 算法的思维方式与人的智能思维方式和推理方式有很多相通之处，可谓瑕不掩瑜。

站在整棵树的顶端往下看，一个远处的大的奖励值的获取，是通过一系列步骤得到的。通过这种反向推理，可以推断出近处的这些动作和状态哪些是好的、哪些是不好的，以及哪些状态的估值是与其当前的瞬时奖励值严重不匹配的。这就是一种量化的智能推理过程——看起来真的很完美。

换一个角度，从一个状态向上，也就是向树的顶端看，能看到什么呢？之所以能够到达当前这个靠谱的状态（可以获得这么大的一个奖励值），是因为当前状态的前一个状态和前一个动作靠谱。而站在前一个被我们推断为靠谱的状态继续向上看，又是这样一个逻辑——前前一个状态和前前一个动作靠谱。这种反向推理过程，像不像我们在做了一些事情之后进行的总结和复盘的过

程呢？我觉得是类似的。

Q-Learning 是一种科学地量化推理过程的算法体现。Q-Learning 算法的思想，在强化学习及深度强化学习领域有巨大的影响力。

2. 强化学习算法的发展

从研究 MDP 时使用的值迭代法，到蒙特卡罗法，再到时间差分法的 SARSA 和 Q-Learning，算法的发展也是有章可循的。

作为值迭代法的一种，动态规划法是依靠类似递归的方式自下而上对状态进行估值计算的。因为是 MDP 的环境，所以相对比较容易计算瞬时回报的数学期望值，也就是说，相对比较容易算出估值。如果环境不满足 MDP，就用蒙特卡罗法，在完整地完成一个 Episode 后，对一个状态进行估值计算。后者的适用范围更广，不需要模型，但方差比较大，且必须在一个 Episode 结束后才能算出估值。另外，使用时间差分法，可以在完全不知道模型状态的情况下，一边尝试收集 Transition 信息，一边计算估值——立等可取。这些方法的普适性一个比一个好。

看懂了这些内容之后，你就不会觉得强化学习有多难了，因为它几乎就是为了进行树搜索而做的各种准备工作。准备工作一旦完成，自然会得到高质量的策略。

3. 强化学习算法面临的挑战

随着时间的推移，计算机的计算能力不断增强，存储设备（空间）的价格不断降低，人们不再满足于用计算机处理几十维、几百维的状态数据。能不能输入完整的屏幕信息？能不能处理视频流？能不能处理几十种动作输入的情形？能不能处理多个智能体协同工作的场景？这些问题，对每一个传统的强化学习算法来说，都是极其艰巨的挑战。

如果使用 Q-Learning 这样的算法来处理高维数据，将是一场灾难。例如，一个游戏环境的输入是 400 像素 × 300 像素的图像，那么表示一个状态的维度至少有 120000 个。在这么多的维度中，很可能包含大量的无用信息（确实不是每一个像素都向我们提供了有用的内容）。

要解决这个问题，有两种方法。

一种方法是，由人来做特征提取，在降维后用 Q-Learning 算法进行训练。不过，特征提取是一件非常艰难的事情，因为要构造一个函数 $f(x|\theta)$，输入一幅 120000 维的图片，输出一个包括数十维（或者百来维）的有用的特征值……说实话，真没有特别靠谱的办法。

另一种方法是，不提取特征，改为动用更高级的人工智能手段，利用 CPU 和 GPU 的计算优势，以及样本获取成本低廉的优势，自动完成这样的工作。

如果你接触过深度学习，此时，你可能会把目光转向它，因为它的优势已经在很多极为复杂的人工智能领域得到了验证。是的，我们马上就要开始接触与深度学习有关的内容，享受深度学习为强化学习的应用和落地提供的诸多便利了。

5.6　本章小结

对传统强化学习的讨论，先告一个段落。绝大多数的传统强化学习方法，从本质上看，还是在解决类似树搜索的问题。

一个好的策略，要有好的表现，要走得远，要能获得高回报。那么，该如何获得一个好的策略呢？可以穷举一棵树吗？不是不可以，但如果这棵树“枝繁叶茂”，就算开了“上帝视角”，也要在每次做决策时进行复杂的搜索——成本有点高。所以，我们还是希望能通过一种只搜索一层的方法，在一个状态下直接选择价值最大的那个动作。那么问题来了：价值并不是瞬时回报值，它无法简单地获取，而需要通过计算来获取。在得到准确的估值后，就可以用直接广度优先遍历树的方法来解决——只“遍历”一层就可以了。

如果环境满足 MDP，那么针对一个状态的输入，我们知道下一个状态的转移概率，因此，只需要知道 $r = R(a|s)$，就可以通过计算动作分布（动作输出的概率及其比例）来计算下一个状态的奖励值的数学期望值了。虽然回报不仅要看短时间的瞬时回报，还要看很多步，但是，从一棵完整的树的底部，自下而上一层一层地计算并更新，肯定可以把一个状态的估值 $V(s)$ 和一个状态下动作的估值 $Q(s,a)$ 估算准确。这就是动态规划法的思路。

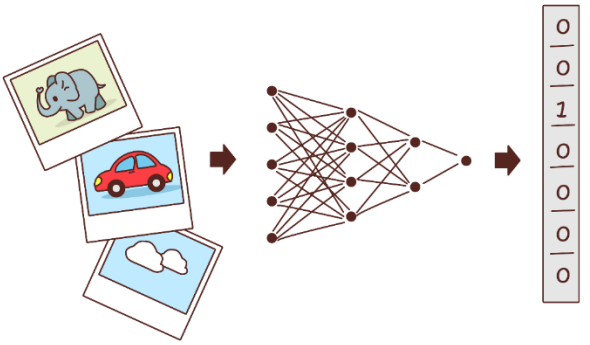
如果环境不满足 MDP，或者我们不知道这个模型是什么样子的，那也没关系，我们还有蒙特卡罗法。多次在环境中做实验，在每个 Episode 结束后估算这个 Episode 上各个状态的估值 G_t，再通过多次实验取平均值的方法得到这个状态的估值——这种方法的普适性更强。

如果希望估值返回得更快、更直接，也是可以实现的。用时间差分法，不必等一个 Episode 结束就可以返回估值，走一步就进行一次学习（这也会使得估值的更新更为平滑）。因为每走一步都需要更新估值，所以，这种方法其实比较希望每一步返回的瞬时奖励值是一个比较“可信”的奖励值，是一个能将我们引导到正确方向上的奖励值。

在这个数据爆炸、维度爆炸的时代，传统的方法已经不能满足工程人员的需求了，因此，需要挖掘新的方法作为补充。深度学习就是在这样一个时间节点崛起的，并给我们带来了意想不到的收获。可以说，正因为有了深度学习工具，强化学习才能进一步展现自己的魅力。

接下来，让我们一起感受深度学习的力量，以及深度学习与强化学习的结合所产生的巨大能量吧。

现代篇



第 6 章　深度学习

在“传统篇”中，我们已经对强化学习中最经典的问题及算法有了相应的了解。可以说，现在你已经可以上手做一些简单的强化学习训练工作了。因为这本书的主要内容是深度强化学习的落地，所以在工程代码方面主要提供的是深度强化学习的应用案例。至于对思维脉络的了解，“传统篇”的内容已经足够了。

接下来，我们就要开启本书最长的一章内容了——深度学习（Deep Learning）。这一章的内容超过了 100 页，对深度学习涉及的知识进行了浓缩和必要的铺垫。这一章的内容，能够帮助我们了解深度学习、在深度强化学习的范畴中应用深度学习的思想与技术。

为什么要将深度学习和强化学习结合起来呢？在那些相对较为复杂的环境中，有关传统强化学习的知识在实际应用中可能不足以解决问题。随着场景的复杂化，尤其是在期望以输入图像为 State 的场景中（希望机器人只是看看屏幕、看看奖励值，就能自己完成一系列进化工作），描述 State 所使用的维度也需要相应增加。

这样的场景和我们在“传统篇”中讨论的场景有一些不同。在“传统篇”中，我们讨论的内容主要是：如何靠谱地估算一个状态的价值或者一个状态下一个动作的价值，如何通过贝尔曼方程对奖励值进行有效回溯，以及如何通过统计学的手段（例如，多次测量取平均值）来逐步把这个价值估算准确。可以说，这一系列问题都是在处理低维度的 State 数据问题，也可以说，只是在理论上讨论这样的问题应该如何解决。既然如此，对于维度极其有限的比大小、搜索等操作来说，难度不大，也无须专门作为一个课题去讨论。

然而，在实际工作中，由于我们期望的 State 输入较为复杂，例如一幅图片、一个连续的视频流（希望机器人自己看看屏幕就能解决问题），如果再使用传统的机器学习或者统计学方法，就有些捉襟见肘了。这个时候，需要一些功能更为强大的解决方案加入进来，帮我们解决相关的搜索问题（至少在一定程度上解决相关的问题）。于是，科学家们自然而然想到了 Deep Learning——结合深度学习的方法解决这一问题。

深度学习主要是指基于深度神经网络的机器学习解决方案。基于深度神经网络的优势，在使用了深度学习的工程作业中，模型表现出了过人的解决问题的能力。

1. 较强的对分类和回归问题的拟合能力

因为深度神经网络有着较高的 VC 维，所以它在分类模型的拟合能力及回归问题的拟合能力

上有着非常好的表现。在传统机器学习中比较难解决的问题，在深度学习中就容易得多（当然，同时需要使用更多的样本和复杂的正则化技巧来解决过拟合问题）。

2. 较高的泛化能力

深度神经网络中有一些非常重要的结构单元，例如卷积核（Convolution Kernel），也称卷积层（Convolutional Layer）。

卷积核的好处显而易见。它可以作为特征提取的工具，相对于全连接网络来说，能够有效降低过拟合的危害、提高泛化性、减少参数的数量。还有池化层（Pooling Layer），这种层可以对输入的图像（或者特征图）进行降采样（Subsampling）处理，也就是缩小它的尺寸，从而在一定程度上继续提高泛化性、减少参数的数量。

因此，深度学习与强化学习的结合，给了我们更多进行 End-to-End[①]训练的空间。下面我们就讲解深度学习的基本原理和应用过程，为后面将要开展的实验做铺垫。为了学好这一章的内容，建议大家安装 PyTorch 并下载本书的示例代码，根据示例代码的内容逐步熟悉基本概念。

6.1　PyTorch 简介

现在，本书的另一个主角出现了，那就是 PyTorch。在正式开始介绍深度学习的各种模型和注意事项之前，我们有必要认识一下 PyTorch。

PyTorch 是一个优秀的开源机器学习和深度学习算法框架，官方网站为 https://pytorch.org/，如图 6-1 所示。

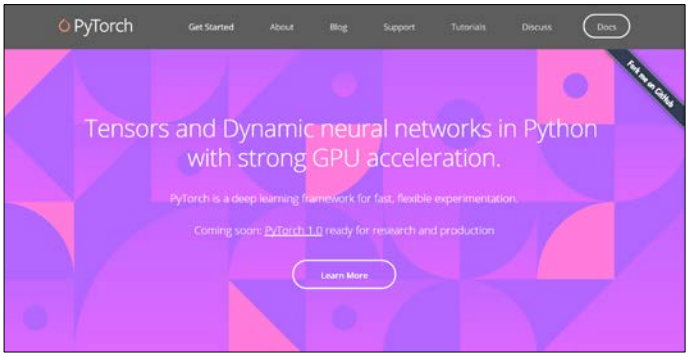


图 6-1　PyTorch 官方网站

PyTorch 的官方网站页面清爽、简洁，提供了 TensorFlow、MXNet、CNTK 等主流的深度学习框架。

① 通常指从原始的输入到原始的输出，其间训练者不再进行任何特殊的加工。例如，在针对游戏的强化学习中，输入为游戏画面，输出为动作。

6.1.1 历史渊源

PyTorch 是一个基于 Torch 框架的 Python 语言机器学习开源库。Torch 是一个历史悠久的机器学习算法库，主要支持 Lua 接口语言。在 Torch 上面有不少优秀的开源深度学习应用，著名的 Neuraltalk2（https://github.com/karpathy/neuraltalk2）就是基于 Torch 和 Lua 编写的，如图 6-2 所示。

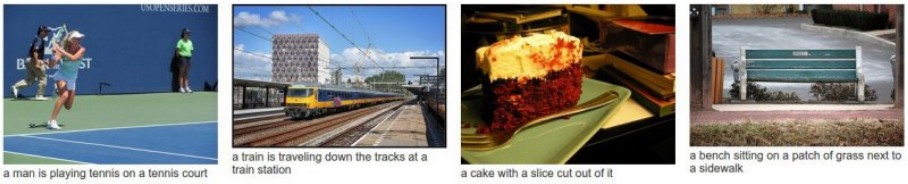


图 6-2 基于 Torch 的 Neuraltalk2

这个深度学习应用非常有趣，而且很有人工智能“范儿”，能够根据一张输入的图片产生一段描述文字——举这个例子主要是为了说明一直以来 Torch 还是小有名气的。

由于 Python 语言“随心所欲”的个性所带来的行业冲击力，除了计算机行业的从业者，几乎任何行业的从业者想要涉足机器学习领域，都会优先选择学习 Python。为了适应 Python 语言越来越热门的趋势，Facebook 在 2017 年 3 月对 Torch 进行了一次比较大的包装和升级，提供了对 Python 语言的深度支持——PyTorch 发布了它的 Beta 版本。

2018 年 4 月，另一个重要的深度学习框架 Caffe2[①]也宣布将代码并入 PyTorch，并不断迭代。可以说，PyTorch 将越来越流行的趋势是非常明显的。如果没有特殊的外力作用，我估计，以后市面上很可能就只剩下 TensorFlow 和 PyTorch 两个主流的深度学习框架了。

TensorFlow 的优势非常明显。TensorFlow 背后是强大的谷歌技术社区，这样的开源生态会提供比较好的发展环境，会有散布在全球的“好事”者给代码“挑毛病”、编写升级包甚至开发第三方封装接口等。因此，在 GitHub 上很容易就能找到以 TensorFlow 为框架的落地工程应用。此外，TensorFlow 推出了针对移动设备环境的 TensorFlow-Lite 框架，使其工业应用能力得到了进一步提升。这些都是目前 PyTorch 无法与 TensorFlow 相比的。

PyTorch 也有很多自己的优势。PyTorch 代码简洁、直观，而且没有太多复杂的由第三方封装的接口，甚至在底层实现上也比 TensorFlow“轻”——至少在同样的硬件环境和神经网络拓扑中，我几乎每次都能感觉到 PyTorch 的运行速度要比 TensorFlow 快一些。我相信，PyTorch 有着非常好的发展前景，毕竟 TensorFlow、PyTorch、Theano、Chainer、Keras、Caffe 等深度学习框架经过角逐，人气最旺的是 TensorFlow 和 PyTorch。顺便说一句：如果不是因为 Caffe 的安装太烦琐，我还是很爱它的……

① 一个著名的深度学习框架，作者为人工智能专家贾扬清。

6.1.2　支持

为了帮助广大深度学习爱好者尽快学会这种框架的应用，PyTorch 在普及方面的努力可谓不遗余力。除了本身在语法设计上尽可能地简洁、优美，PyTorch 还提供了很多学习资料，大家可以访问 https://pytorch.org/tutorials/ 下载，如图 6-3 和图 6-4 所示。

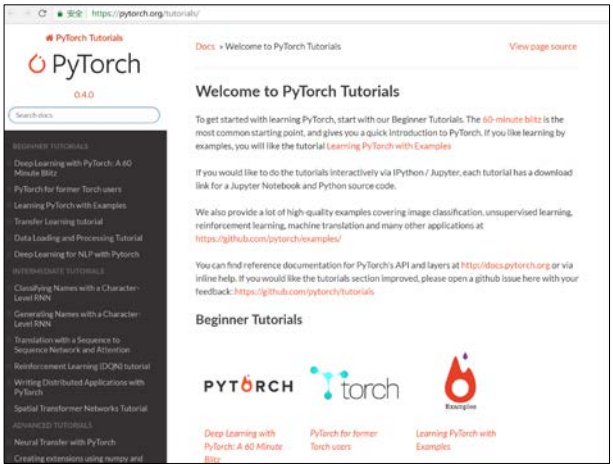


图 6-3　PyTorch 教程

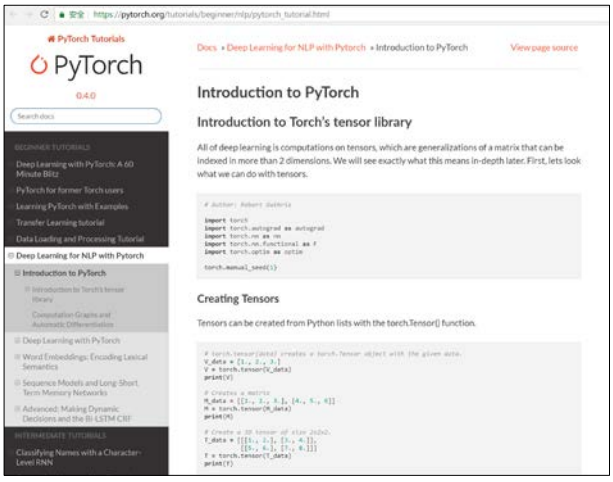


图 6-4　PyTorch 介绍

PyTorch 有大量的代码资源可供学习，只不过在其官方网站上找不到完整的文档。如果想获取相关文档，可以访问 http://pytorch-cn.readthedocs.io/zh/latest/，如图 6-5 所示。

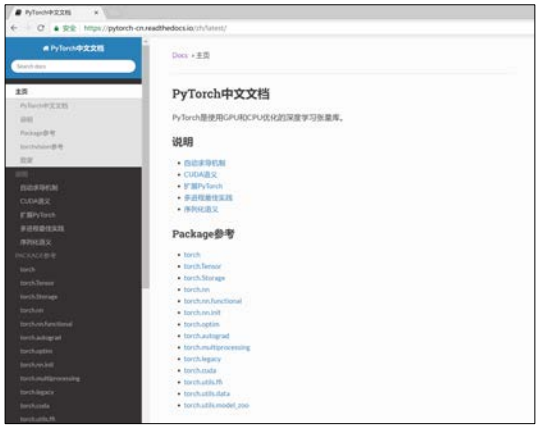


图 6-5　PyTorch 的相关中文文档

在开源氛围的感染下，总是有很多贡献者愿意提供有益的代码、文档等有价值的精神产物。我们还可以在这个网站上找到其他开源软件的中文文档，当然，其中的大多数都是从其英文技术手册翻译过来的。这个网站带有博客的性质，其中的文档都来自第三方，质量难免良莠不齐，所以，如果要参考其中的文档，恐怕免不了要自己甄选一番。

如图 6-6 所示，PyTorch 主要的包就这么多，看起来也比较清楚——起码比 TensorFlow 清楚得多。相信用过 TensorFlow 的读者朋友都知道，由于多方“混战”，在 TensorFlow 里“一个网络，多种写法”的情况屡屡发生，除了原生的 tf.nn 包，还有 slim、keras 等提供了很多声明网络组建写法的包。说实在的，在一个只有几十户人家的小区中，住户们用三四种语言进行交流，不是一件很愉快的事情。

图 6-6　PyTorch 中主要的包

如果想对 PyTorch 的底层环境进行细致的研究，甚至想从源代码层面对其进行修改，就免不了要参考 GitHub 上开源的完整源代码（https://github.com/pytorch/pytorch），如图 6-7 所示。

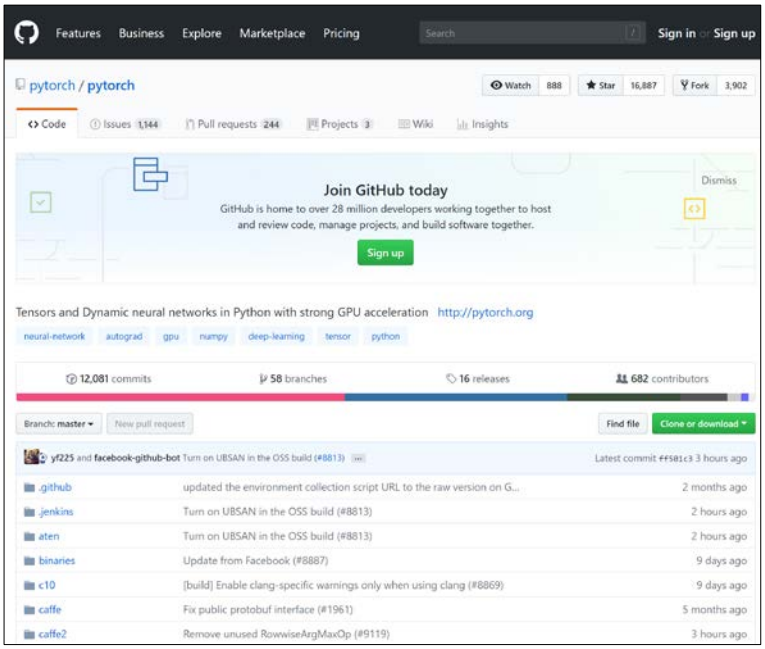


图 6-7　GitHub 上 PyTorch 的主页

从势头上看，PyTorch 锐不可当，在写作本书时已经获得了近 1.7 万颗星。不过，PyTorch 要想追上 TensorFlow（超过 10 万颗星，如图 6-8 所示），还需要一些时间。从代码文件的命名来看，PyTorch 和 TensorFlow 的大部分代码都是用 C 或 C++ 编写的，毕竟为了尽可能保持软件运行的高效性，还是要使用这些执行效率比较高的语言。

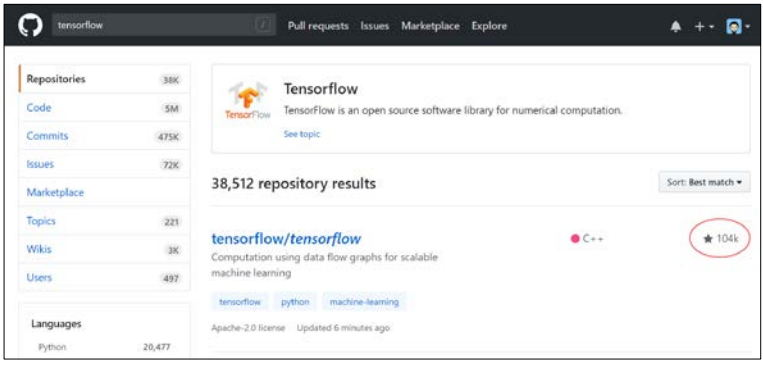


图 6-8　TensorFlow 在 GitHub 上获得的星

PyTorch 从安装到使用都比较简单。具体安装过程，可以参考本书的附录 A，也可以自行参考网上的文档。使用 pip 方式安装 PyTorch，一条命令就够了，非常方便。

必须承认的是，开源软件的生态环境有了很大的改善，越来越多的开源软件之间加强了支持与合作。例如，从前要在 Ubuntu 14.04 上安装一个支持 CUDA 的 TensorFlow 或者 Caffe 环境，不花上一整天时间是无法完成的。在那个时候，操作系统也好，各种开源框架也罢，甚至包括英伟达的 CUDA 驱动，都处于“各自为战”“自说自话”的状态，而这让使用者痛苦非常。有“Linux 之父”之称的著名开源软件先驱林纳斯·托瓦兹[①]，在 2012 年 6 月接受电视采访，谈到英伟达公司的显卡对 Linux 操作系统的支持不够友好的时候，就当着现场观众的面，对英伟达公司的做法表示了愤慨。

关于这件事情，我认为没有必要苛责林纳斯·托瓦兹和英伟达公司两者中的任何一个，尤其是英伟达公司。技术环境生态的成熟需要一个过程，在这个过程中，有大势所趋的发展方向，有在具体时间节点上最应该被释放出来的具体事件。每件事都有发生的时间点，如果当时英伟达对 Linux 的支持不够完善（或者说不够积极），那么经过这些年的发展，我想，英伟达再也没有理由不对 Linux 提供良好的支持了——至少在 Ubuntu 17.10 上安装 CUDA 驱动，十几分钟就完成。这些“基础设施”的建设与完善，自会有专业的公司来实现，而我们应该更多地关注模型和业务。

至于 PyTorch 的使用方法，我们会在后面的章节中根据应用具体讲解。

6.2 神经元

说到深度学习，就必须先说说神经网络。

神经网络也称作人工神经网络（Artificial Neural Network，ANN）。我们现在所说的深度学习就是基于深度神经网络实现的一系列应用的统称。神经网络是人类受到生物神经细胞结构的启发而研究出来的一种算法与建模体系。

人的神经元结构，如图 6-9 所示，枝杈很多，远远看去，一边比较粗大，另一边比较纤细。粗大的部分是细胞体。细胞体上有一些小枝杈，叫作“树突”。细胞体下方像尾巴一样细长的东西叫作“轴突”。不同细胞之间通过树突和轴突传递信息，它们的接触点叫作“突触”。准确地说，一个细胞的轴突通过突触将信号传递给另一个细胞的树突。

神经细胞使用化学信号进行信息的传递。化学信号依靠一些化学分子的传输来传递信息。尽管化学分子非常复杂，人类对化学分子所承载的信息一知半解，没有形成体系性的认识，但是，人类仍从神经细胞通过彼此刺激来传递信息的方式中获得了启迪——是否可以设计一种呈网状连接的处理单元，让它们通过某种方式互相刺激，协同完成信息处理工作呢？

① Linus Torvalds（1969 年—），著名的计算机程序员，Linux 内核的发明人及该计划的合作者。

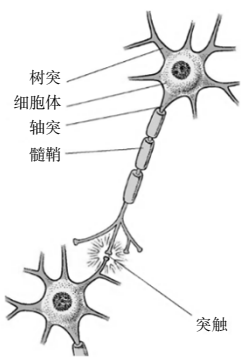


图 6-9　神经元的结构

1957 年 Rosenblatt[①]提出的感知器模型（Perceptron）是较早出现的神经网络模型。这种模型和较新的应用框架中的神经网络单元在形式上是非常接近的。要想了解神经网络，就应该先了解它最基本的组成单元——神经元。我们之前接触的算法，大都用于完成四则运算（加、减、乘、除）、比大小、循环、分支、读写数据等，即便目标任务比较复杂，通过这些基本的组合也能实现。然而，神经网络和这种方式还真有那么点不一样。我们先来看看神经元是什么。

如图 6-10 所示，一个最简单的神经元，有一个输入和一个输出，所以它所表达的含义和一个普通的函数没有任何区别。不过请注意，现在我们使用的神经元通常由两部分组成，一是“线性模型”部分，二是“激励函数”部分。如果你对机器学习的相关知识了解不多，就要先认识一下“线性模型”。

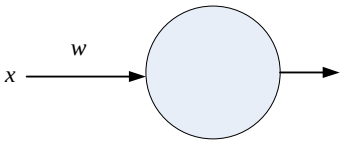


图 6-10　一维输入的神经元结构

假设这个神经元的函数表达为

$$f(x) = x + 1,$$

那么这就是一个普通的一次函数。也就是说：当输入为 $x = 1$ 时，输出端的 $f(x) = 2$；当输入为 $x = 100$ 时，输出端的 $f(x) = 101$。

怎么样，是不是很简单？没错，这就是一个最简单的神经元的工作原理。也许你要说，这么简单的东西为什么非要往神经元上靠，这不是把简单的问题复杂化了吗？对这个问题，我暂时不

① Frank Rosenblatt，（1928 年 7 月—1971 年 7 月），美国著名心理学家，人工智能科学家。

作解释。我们再看一个例子，如图 6-11 所示。

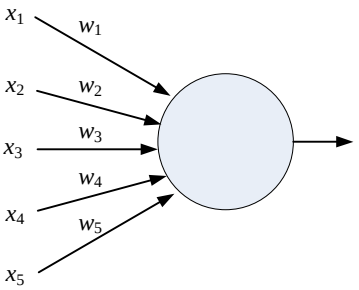


图 6-11　五维输入的神经元结构

$\boldsymbol{x}$ 既然可以是一个一维的向量（如图 6-10 所示），就可以是一个五维的向量（如图 6-11 所示），也可以是一百维或者更多——我们就说它的输入为一个 n 维的向量吧。按照一维向量的处理方式，我们可以建立一个有 n 个输入项的神经元 $f(\boldsymbol{x})$，展开写就是 $f(x_1, x_2, \ldots, x_n)$。在这里特别声明一下，这个 $\boldsymbol{x}$ 是一个 n 维的向量，带有一个输出函数值 output，也就是说，output 的值是函数的输出值 $f(x)$，即 output $= f(x)$。这个函数的处理过程写作

$$f(\boldsymbol{x}) = w\boldsymbol{x} + b.$$

这也是神经元的核心部分对 $\boldsymbol{x}$ 所做的线性处理。其中，$\boldsymbol{x}$ 是一个 $n \times 1$ 的矩阵，$\boldsymbol{w}$ 是一个 $1 \times n$ 的权重矩阵，b 是偏置项。虽然这种写法看上去很简洁，但容易让初学者犯糊涂——至少没看明白 $\boldsymbol{w}$ 和 $\boldsymbol{x}$ 还有 b 一起做了什么。我们把这个计算过程拆解一下。例如，

$$\boldsymbol{w} = [0 \quad 1 \quad 2 \quad 3 \quad 4],$$

$$\boldsymbol{x} = \begin{bmatrix} 5 \\ 4 \\ 3 \\ 2 \\ 1 \end{bmatrix},$$

$$b = 2.$$

那么，此时 y 等于几呢？

$$y = [0 \quad 1 \quad 2 \quad 3 \quad 4] \begin{bmatrix} 5 \\ 4 \\ 3 \\ 2 \\ 1 \end{bmatrix} + 2 = (0 \times 5 + 1 \times 4 + 2 \times 3 + 3 \times 2 + 4 \times 1) + 2 = 27.$$

$\boldsymbol{wx}$ 就是指一个 1 行 n 列和一个 n 行 1 列的向量相乘，即对应维度的值相乘（在这个例子中，$n = 5$）——就这么简单。这种计算方式叫作点积，是矩阵运算中最简单的一种。

6.3　线性回归

了解了神经元的工作方式，我们就可以聊聊线性回归了。线性回归也是众多机器学习问题中非常简单的一个。然而，越简单的内容就越基础，越基础的内容就越重要。下面，我们以线性回归为例，迈过 PyTorch 大门外最低的那道槛。

打开本书的代码文件夹，进入 01-basics/linear_regression 目录。

打开文件 main.py，可以看到如下内容。

```
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
from torch.autograd import Variable


# Hyper Parameters
input_size = 1
output_size = 1
num_epochs = 600
learning_rate = 0.001

# Toy Dataset
x_train = np.array([[3.3], [4.4], [5.5], [6.71], [6.93], [4.168],
                    [9.779], [6.182], [7.59], [2.167], [7.042],
                    [10.791], [5.313], [7.997], [3.1]], dtype=np.float32)

y_train = np.array([[1.7], [2.76], [2.09], [3.19], [1.694], [1.573],
                    [3.366], [2.596], [2.53], [1.221], [2.827],
                    [3.465], [1.65], [2.904], [1.3]], dtype=np.float32)

# Linear Regression Model
class LinearRegression(nn.Module):
    def __init__(self, input_size, output_size):
        super(LinearRegression, self).__init__()
        self.linear = nn.Linear(input_size, output_size)

    def forward(self, x):
        out = self.linear(x)
        return out

model = LinearRegression(input_size, output_size)

# Loss and Optimizer
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

# Train the Model
```

```
for epoch in range(num_epochs):
    # Convert numpy array to torch Variable
    inputs = Variable(torch.from_numpy(x_train))
    targets = Variable(torch.from_numpy(y_train))

    # Forward + Backward + Optimize
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = criterion(outputs, targets)
    loss.backward()
    optimizer.step()

    if (epoch+1) % 5 == 0:
        print ('Epoch [%d/%d], Loss: %.4f'
              %(epoch+1, num_epochs, loss.data[0]))

# Plot the graph
predicted = model(Variable(torch.from_numpy(x_train))).data.numpy()
plt.plot(x_train, y_train, 'ro', label='Original data')
plt.plot(x_train, predicted, label='Fitted line')
plt.legend()
plt.show()

print(model.state_dict())

# Save the Model
torch.save(model.state_dict(), 'model.pkl')
```

main.py 文件中只有 66 行代码，但完成了一个完整的训练过程。

如果使用 `python main.py` 命令执行这个文件，会得到如图 6-12 所示的线性回归图。图中的点表示样本点，线是拟合出来的，也可以说，我们观测到的是图中的点，而通过学习得到的是 $y = wx + b$ 这条线。在这个例子中，w 和 x 都是一维的，这与我们在初中时画过无数次的 $y = kx + b$ 所对应的线没有区别。在那时，k 叫作斜率，b 叫作截距。如果非要说有区别，那仅仅在于：在初中的时候，我们知道 k 和 b 指的是什么，而现在的情况反过来了，我们知道每个点所对应的 x 和 y，要求 w 和 b 取什么值。也就是说，已知坐标 (x, y)，但不知道 x 和 y 之间的对应关系。

通过观察可以发现，貌似 x 和 y 之间有一个关系 $y = wx + b$，但不知道其中的 w 和 b 分别为多少。于是，我们可以调用 PyTorch 的 nn.Linear 进行训练，通过类似解方程的方式求出 w 和 b。最后，根据求出来的 w 和 b，画出 $y = wx + b$ 这条线。

整个过程就是这样。

可能你要问：怎么通过解方程的方法把这个值求出来呢？计算机是怎么做到的呢？别着急，自然是有办法的。

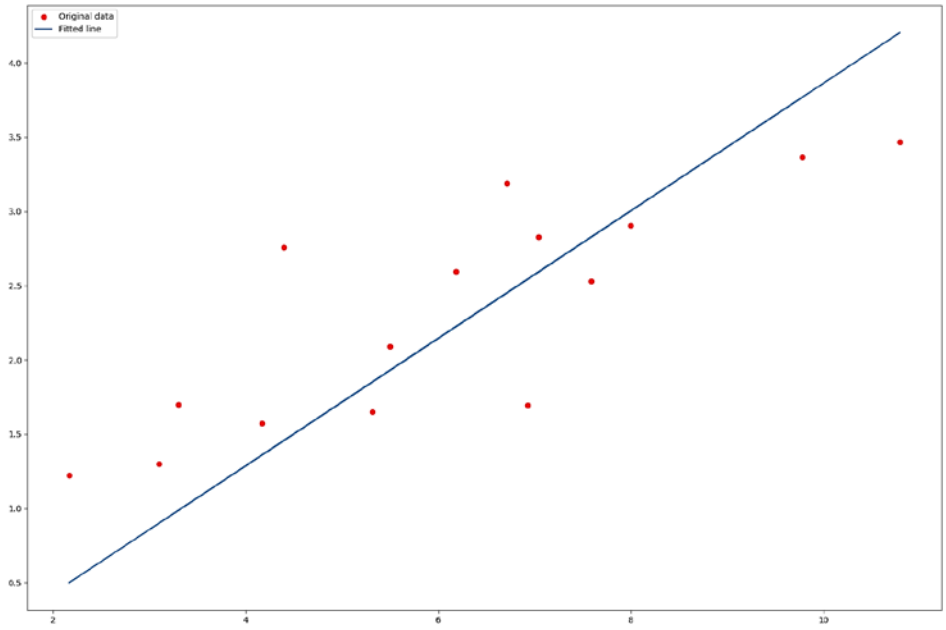


图 6-12　线性回归

我们浏览一下这部分代码，看看 PyTorch 的“文风”怎么样。

- 1 到 5 行：文件的包导入部分。
- 8 到 12 行：参数设定部分。
- 14 到 21 行：进行数据集赋值。数据集一般都是从文件或者数据库中读取的，在这里为了操作方便，以硬编码[①]（Hard Coding）的形式将其写入。
- 23 到 31 行：定义线性回归函数 $y = wx + b$。这里写出来是 `nn.Linear`，它是使用 PyTorch 的 nn 包中的 `Linear` 类实现的。
- 33 行：对封装的 `LinearRegression` 类（也就是 PyTorch 自带的线性回归类）进行实例化。
- 36 行：定义损失函数 `nn.MSELoss`（也就是均方损失函数，参见 6.6.4 节）。
- 37 行：定义优化器。在这里，使用随机梯度下降（Stochastic Gradient Descent，SGD）的方法对损失函数进行优化，也就是让计算机使用 SGD 方法来更新待定系数，让它向着减小损失函数的值的方向变化。
- 40 到 54 行：训练模型。
- 56 到 61 行：绘图。
- 64 行：保存模型文件。

① 也可以写作“Hard Code”“Hardcode”等，是指在代码编写过程中，将一些本应作为参数的内容（例如路径名、文件名、阈值等）直接以不可变的方式写在代码中。在通常情况下，硬编码不是一种值得提倡的编程风格。

从形式上看，确实非常简单，对不对？如果你在阅读这段代码的过程中遇到了一些不太明白的词汇，也不必着急，后面会分别介绍它们的用途。仅用 66 行代码就能完成一个类似解方程、解模型的任务，是不是很酷？是的，一个神经元——准确地说是半个神经元，也就是一个神经元的前半部分——已经可以作为一个独立的模型来工作了。我们完全可以把这个线性回归的例子看成一个神经元独立工作的产物。为什么说是半个神经元呢？另外半个神经元是什么呢？另外半个神经元通常是指激励函数。

6.4 激励函数

激励函数（Activation Function，也译作“激活函数”）是神经元的重要组成部分。

在一个神经元中，激励函数跟在 $f(x)=wx+b$ 函数后面，用于添加一些非线性的因素，如图 6-13 所示。

Name	Plot	Equation	Derivative (with respect to x)	Range	Order of continuity	Monotonic	Derivative Monotonic	Approximates identity near the origin
Identity		$f(x)=x$	$f'(x)=1$	$(-\infty,\infty)$	C^∞	Yes	Yes	Yes
Binary step		$f(x)=\begin{cases}0 & \text{for } x<0\\1 & \text{for } x\geq 0\end{cases}$	$f'(x)=\begin{cases}0 & \text{for } x\neq 0\\? & \text{for } x=0\end{cases}$	$\{0,1\}$	C^{-1}	Yes	No	No
Logistic (a.k.a. Soft step)		$f(x)=\frac{1}{1+e^{-x}}$	$f'(x)=f(x)(1-f(x))$	$(0,1)$	C^∞	Yes	No	No
TanH		$f(x)=\tanh(x)=\frac{2}{1+e^{-2x}}-1$	$f'(x)=1-f(x)^2$	$(-1,1)$	C^∞	Yes	No	Yes
ArcTan		$f(x)=\tan^{-1}(x)$	$f'(x)=\frac{1}{x^2+1}$	$\left(-\frac{\pi}{2},\frac{\pi}{2}\right)$	C^∞	Yes	No	Yes
Softsign [7][8]		$f(x)=\frac{x}{1+\lvert x\rvert}$	$f'(x)=\frac{1}{(1+\lvert x\rvert)^2}$	$(-1,1)$	C^1	Yes	No	Yes
Rectified linear unit (ReLU) [9]		$f(x)=\begin{cases}0 & \text{for } x<0\\x & \text{for } x\geq 0\end{cases}$	$f'(x)=\begin{cases}0 & \text{for } x<0\\1 & \text{for } x\geq 0\end{cases}$	$[0,\infty)$	C^0	Yes	Yes	No
Leaky rectified linear unit (Leaky ReLU) [10]		$f(x)=\begin{cases}0.01x & \text{for } x<0\\x & \text{for } x\geq 0\end{cases}$	$f'(x)=\begin{cases}0.01 & \text{for } x<0\\1 & \text{for } x\geq 0\end{cases}$	$(-\infty,\infty)$	C^0	Yes	Yes	No
Parameteric rectified linear unit (PReLU) [11]		$f(\alpha,x)=\begin{cases}\alpha x & \text{for } x<0\\x & \text{for } x\geq 0\end{cases}$	$f'(\alpha,x)=\begin{cases}\alpha & \text{for } x<0\\1 & \text{for } x\geq 0\end{cases}$	$(-\infty,\infty)$	C^0	Yes iff $\alpha\geq 0$	Yes	Yes iff $\alpha=1$
Randomized leaky rectified linear unit (RReLU) [12]		$f(\alpha,x)=\begin{cases}\alpha x & \text{for } x<0\\x & \text{for } x\geq 0\end{cases}$ [1]	$f'(\alpha,x)=\begin{cases}\alpha & \text{for } x<0\\1 & \text{for } x\geq 0\end{cases}$	$(-\infty,\infty)$	C^0	Yes	Yes	No
Exponential linear unit (ELU) [13]		$f(\alpha,x)=\begin{cases}\alpha(e^x-1) & \text{for } x<0\\x & \text{for } x\geq 0\end{cases}$	$f'(\alpha,x)=\begin{cases}f(x)+\alpha & \text{for } x<0\\1 & \text{for } x\geq 0\end{cases}$	$(-\alpha,\infty)$	C^1 when $\alpha=1$, otherwise C^0	Yes iff $\alpha\geq 0$	Yes iff $0\leq\alpha\leq 1$	Yes iff $\alpha=1$
S-shaped rectified linear activation unit (SReLU) [14]		$f_{t_l,a_l,t_r,a_r}(x)=\begin{cases}t_l+a_l(x-t_l) & \text{for } x\leq t_l\\x & \text{for } t_l<x<t_r\\t_r+a_r(x-t_r) & \text{for } x\geq t_r\end{cases}$ t_l,a_l,t_r,a_r are parameters.	$f'_{t_l,a_l,t_r,a_r}(x)=\begin{cases}a_l & \text{for } x\leq t_l\\1 & \text{for } t_l<x<t_r\\a_r & \text{for } x\geq t_r\end{cases}$	$(-\infty,\infty)$	C^0	No	No	No
Adaptive piecewise linear (APL) [15]		$f(x)=\max(0,x)+\sum_{s=1}^{S}a_s^i\max(0,-x+b_s^i)$	$f'(x)=H(x)-\sum_{s=1}^{S}a_s^iH(-x+b_s^i)$ [2]	$(-\infty,\infty)$	C^0	No	No	No
SoftPlus [16]		$f(x)=\ln(1+e^x)$	$f'(x)=\frac{1}{1+e^{-x}}$	$(0,\infty)$	C^∞	Yes	Yes	No
Bent identity		$f(x)=\frac{\sqrt{x^2+1}-1}{2}+x$	$f'(x)=\frac{x}{2\sqrt{x^2+1}}+1$	$(-\infty,\infty)$	C^∞	Yes	Yes	Yes
SoftExponential [17]		$f(\alpha,x)=\begin{cases}-\frac{\ln(1-\alpha(x+\alpha))}{\alpha} & \text{for } \alpha<0\\x & \text{for } \alpha=0\\\frac{e^{\alpha x}-1}{\alpha}+\alpha & \text{for } \alpha>0\end{cases}$	$f'(\alpha,x)=\begin{cases}\frac{1}{1-\alpha(\alpha+x)} & \text{for } \alpha<0\\e^{\alpha x} & \text{for } \alpha\geq 0\end{cases}$	$(-\infty,\infty)$	C^∞	Yes	Yes	Yes iff $\alpha=0$
Sinusoid		$f(x)=\sin(x)$	$f'(x)=\cos(x)$	$[-1,1]$	C^∞	No	No	Yes
Sinc		$f(x)=\begin{cases}1 & \text{for } x=0\\\frac{\sin(x)}{x} & \text{for } x\neq 0\end{cases}$	$f'(x)=\begin{cases}0 & \text{for } x=0\\\frac{\cos(x)}{x}-\frac{\sin(x)}{x^2} & \text{for } x\neq 0\end{cases}$	$[\approx -.217234,1]$	C^∞	No	No	No
Gaussian		$f(x)=e^{-x^2}$	$f'(x)=-2xe^{-x^2}$	$(0,1]$	C^∞	No	No	No

图 6-13 激励函数一览 ①

① 引自维基百科，https://en.wikipedia.org/wiki/Activation_function。

激励函数的种类很多，曲线可谓五花八门。然而，在成熟的深度学习框架中，我们能够使用的激励函数种类有限。下面就介绍常用的激励函数。

6.4.1　Sigmoid 函数

Sigmoid 函数是几乎所有学习神经网络的人接触的第一个激励函数，如图 6-14 所示。它的定义是

$$f(\boldsymbol{x}) = \frac{1}{1 + \mathrm{e}^{-(\boldsymbol{w}\boldsymbol{x}+b)}},$$

也可以写成

$$z = \boldsymbol{w}\boldsymbol{x} + b,\ \ f(z) = \tfrac{1}{1+\mathrm{e}^{-z}}.$$

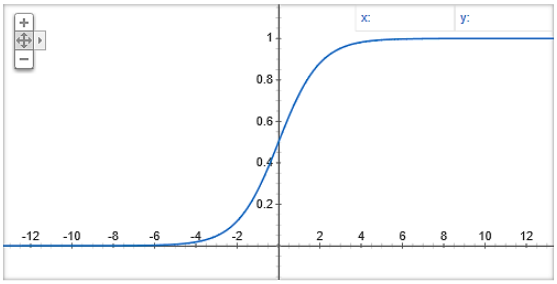


图 6-14　Sigmoid 激励函数

横轴表示 z，纵轴表示 $f(z)$。对一个高维的 $\boldsymbol{x}$ 向量的输入，在 $\boldsymbol{w}$ 和 $\boldsymbol{x}$ 两个矩阵中做完内积（也叫作点积）后加上 b，以这样一个线性模型的结果充当自变量 z，叠加到 $f(z) = \frac{1}{1+\mathrm{e}^{-z}}$ 中。这就使得输入 $\boldsymbol{x}$ 与输出 $f(\boldsymbol{x})$ 的关系与 6.3 节例子中的不同。

在 6.3 节中，我们只讨论了 $f(\boldsymbol{x})$ 以线性回归方式工作的过程，不过那通常不是它在神经网络中工作的状态。当一个完整的神经元被定义时，它通常由“线性模型”和“激励函数”两部分首尾相接而成。这个神经元的前半部分接收外界的向量 $\boldsymbol{x}$（作为刺激），经过线性模型 $\boldsymbol{w}\boldsymbol{x} + b$，以及一个激励函数，得到输出。如图 6-15 所示，为了展示方便，只画出了 6 个 $\boldsymbol{x}$ 的指向线条。在实际工作中，很多全连接网络里的 $\boldsymbol{x}$ 如果完整地画出来，可能有上万条甚至更多。

Sigmoid 函数是一种较早出现的激励函数，把激励值最终投射到了 0 到 1 的区间上。Sigmoid 激励函数通常可以作为概率解释，通过这种方式引入了非线性因素。

为什么要引入非线性因素？这可能是个比较有趣的话题。在用一个大的函数“网络”去拟合一个对应的关系时，如果仅通过线性函数进行拟合，那么拟合的结果一定仅包含线性关系。如果在这个客观的、我们要求解的关系中本应含有非线性关系，那么这个网络必定严重欠拟合（因为一开始设计出来的网络就存在缺陷，所以再怎么训练都不会得到好的结果）。线性就是指用形如 $f(\boldsymbol{x}) = \boldsymbol{w}\boldsymbol{x} + b$ 的表达式表示的输入与输出的关系。其他关系都应该算作非线性关系，在本章后

面会给出具体的例子。

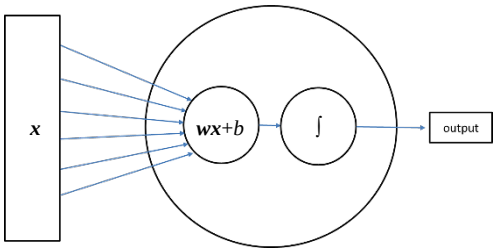


图 6-15 一个带有激励函数的神经元

6.4.2 Tanh 函数

Tanh 函数也算是比较常见的激励函数了，在学习循环神经网络（Recurrent Neural Networks，RNN）的过程中我们会经常见到它。

Tanh 函数也叫作双曲正切函数，它的表达式如下。

$$\mathrm{Tanh}(\boldsymbol{x}) = \frac{\mathrm{e}^{x} - \mathrm{e}^{-x}}{\mathrm{e}^{x} + \mathrm{e}^{-x}}.$$

Tanh 函数的曲线，如图 6-16 所示。

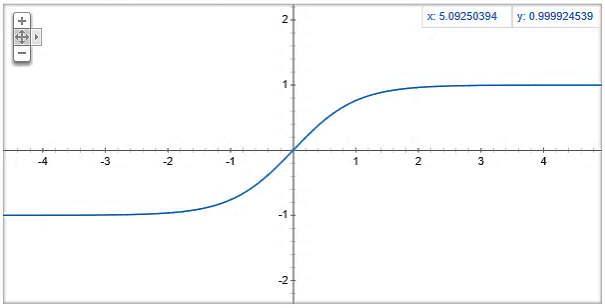


图 6-16 Tanh 激励函数

Tanh 函数和 Sigmoid 函数“长”得很像，都是“S”形的曲线，只不过 Tanh 函数把输入的值投射到 –1 到 1 的区间上了。

6.4.3 ReLU 函数

ReLU 函数是大部分卷积神经网络（Convolutional Neural Networks，CNN）中经常使用的激励函数，它的全称是“Rectified Linear Units”，如图 6-17 所示。

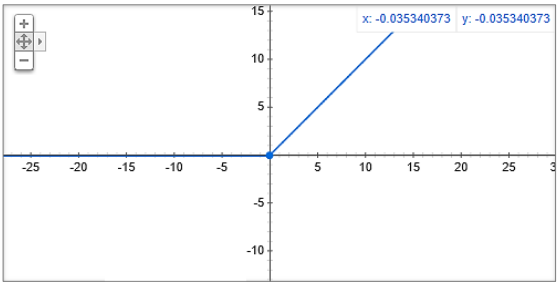


图 6-17　ReLU 激励函数

ReLU 函数的表达式为

$$y = \max(\boldsymbol{x}, 0).$$

函数图象原点左侧部分的斜率为 0，右侧则是一条斜率为 1 的直线。ReLU 函数显然是一个非线性函数，当 x 小于 0 时输出一律为 0，当 x 大于 0 时输出值等于输入值。

6.4.4　Linear 函数

Linear 激励函数在实际应用中使用不多，其原因是：如果网络中前面的线性层引入的是线性关系，后面的激励层还是线性关系，就会让网络无法很好地拟合非线性特性的关系，从而发生严重的欠拟合现象。

Linear 函数的图象如图 6-18 所示，其表达式为

$$f(x) = x.$$

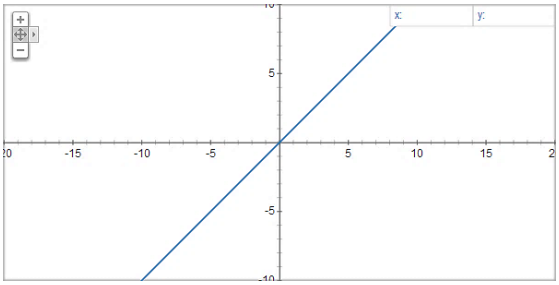


图 6-18　Linear 激励函数

一个函数可以作为激励函数（也可以说，根本没有激励函数），必然是有用武之地的。在什么情况下使用激励函数？如果需要拟合的关系的输出是值域在正负无穷上的连续实数，最后输出的这一层（或者这一环节）还能用 Sigmoid 函数之类的激励函数表示吗？显然不行，因为这类函数的取值范围无法满足我们的需要。正所谓“天生我材必有用”，Linear 激励函数通常会在回归问题的拟合中使用。

6.5 神经网络

一旦多个“神经”首尾连接，形成一个类似网络的结构来协同工作，就可以称之为“神经网络”了。神经网络的层数没有硬性规定。每层网络中有多少个神经元节点，完全是在各个场景中根据经验和一些相关理论进行尝试，获得一个适应当前场景的网络，进而得到的。虽然每年都有不少论文讨论每层网络中有多少个神经元比较合适，但还没有哪个理论具有足够的说服力。

深度学习这个领域就是这么有趣：每年都有人提出新的观点，然后，就有人去附和它、支持它、寻找有利于它的证据；同时，另外一些人会提出相反的观点，或者对其表示悲观，并且很快会聚拢一批有类似观点的人。发展几年后会怎么样？有可能一方被证明为失败的或错误的，然后就此消亡。更多的情况是，两方很可能会分化成两个不同的流派，持有不同的见解，各自完善自己的观点和理论，最后形成两个甚至多个不同的知识体系。这些体系内部各自形成相对独立的完整理论和量化关系，并能比较合理地彼此印证、解释和支撑体系中的各种观点……这是因为，它是前沿的，缺乏完整的、有严格推导和证明的、精确量化的且可以标准化的应用基础。

因此，在学习神经网络（深度学习）的过程中，你会不断遇到不同形式的网络。每种网络，或出自某个具体的工程项目，根据用户的需求及工程人员的经验和实验效果来选定，或出自那些尖端的实验室（例如谷歌、微软等），并辅以相关的论文，对网络的实验效果与同期其他网络的解决方案加以对比。但是，在这些网络的诞生过程中，你极少能发现完整的、严谨的、普适的、不容置疑的推导过程。也难怪有不少多年从事深度学习的资深“老兵”说，“深度学习越学越像老中医”。我觉得这个说法挺形象的——年轻中医没经验，不敢轻易开方子；等熬成了老中医的时候才发现，里面门道太深，深到研究一辈子也没办法总结出完整的、可以精确推导的公式或定理，在大部分情况下只能靠自己的经验和实验结果调整药方的地步。好像挺无奈的，是吧？其实学过就会知道，用好了真能解决问题。我们不太需要“放之四海而皆准”的模型，作为工程技术人员，我们需要的是高度可应验的、有着严格限定条件的落地应用。打个比方，农民不需要了解全世界的气候变化也能种粮食，只要了解自己所在地区的天气规律就够了——显然，后者的难度比较低，也更容易学习和验证。

闲言少叙，咱们来看看神经网络的结构吧。

一个比较简单的神经网络结构，如图 6-19 所示。一个神经网络，通常会分成输入层（Input Layer）、隐藏层（Hidden Layer，也叫隐含层）、输出层（Output Layer）。

输入层直接接收输入的向量，不对数据进行任何处理，所以，这一层通常不计入层数。

隐藏层可以有 1 层，也可以有多层。据我所知，已经出现了超过 150 层的网络（例如著名的深度残差网络 ①ResNet），一些特殊的网络甚至超过了 1000 层！2016 年，商汤科技的深度学习团队就在 ImageNet 图片分类任务中实现了一个 1207 层的网络。先不说这样的网络要怎么训练，仅

① Deep Residual Networks，2015 年问世，可以支持很深的网络，在图片分类和目标识别应用领域表现非常好。

仅是存储这么多层网络里的权重参数，就得消耗不少内存。1207 层已经非常多了，在本书的实验中没有使用这么多层的神经网络。

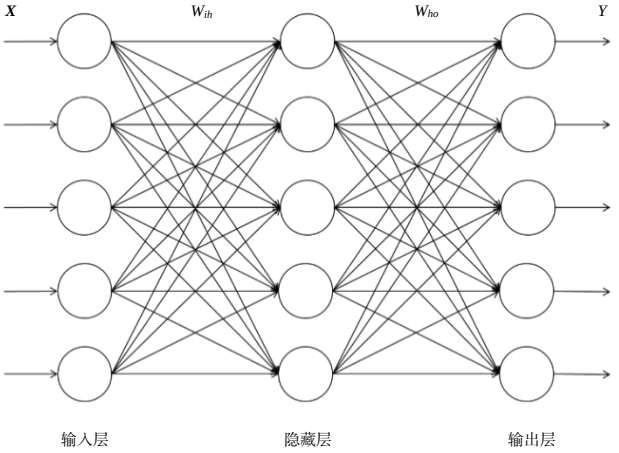


图 6-19　神经网络

输出层是最后一层，用于输出整个网络处理的值。这个值可能是一个分类向量值，也可能是通过类似线性回归那样的方式产生的连续的值，还可能是其他复杂类型的值。例如，一个数字序列或者一段话，根据不同的需求，输出层的构造不尽相同。输出层的定义是直接与需求相关的，你需要它输出什么，就要定义它是什么格式（或者形式）的。

神经元就是通过这种首尾相接的方式进行数据传递的。一个神经元接收数据，经过处理，输出给后一层中相应的 1 个或多个神经元。一个向量中的任何一个维度分量，通过了哪些神经元的处理，以及在输出后又输入了哪些神经元，都被清晰地记录下来了。

6.6　网络训练

网络训练，既困难，又简单。

说它困难，是因为一个完整的训练过程涉及很多基础的数学问题，例如基础数学中的矩阵计算问题、统计学中的回归问题、信息论中的信息熵问题、最优化问题中的梯度下降问题等。如果要深入、细致地把这些问题统统研究清楚，还需要了解相关的工程实现原理，知道常见的“坑”，以及常见问题的解决方案等。可以说，不踏踏实实读到研究生毕业，几乎不可能达到这个程度。这不是危言耸听，尤其当你想在网络训练中进行效率或者精度上的优化时，必须要有深厚的数学功底和工程经验。这不仅需要扎实的知识功底，更需要时间的积累。

说它简单，是因为问题虽然复杂，但有很多是共通的、基础性的问题。这些问题的解决方案，被数据科学家们封装成了一个个组件，只要我们了解这些组件的原理，能够把它们有机地结合起

来就可以了，剩下工作基本上就是调参（对超参数进行尝试和结果对比）了。所以，有人戏称现在的深度学习工程师为“调参工程师”，也不算故意埋汰人。

事实上，这样的方式降低了深度学习的门槛，扩大了深度学习的应用群体。我觉得，这应该算是科学学以致用、走入民间的必由之路吧。想想看，手机、电视机、体感游戏机、电磁炉、微波炉……哪个不是基础数学、物理学、电磁学、通信工程学、半导体制造技术等高、精、尖学科的集合体？这些设备展现给我们的只是一些按钮、开关，即便打开这些设备，里面的各种元件也是高内聚、低耦合的，复杂的组件仍旧被封装在小小的元件内部，元件和元件之间仍旧保持着尽可能简单的连接方式。这可能就是现代科技引领下的超大规模协同制造业的终极解决方案（内部复杂，接口简单），也诠释了高质量的面向对象编程[①]哲学思想指导下的构造精髓。

可以说，所有基于深度学习的概念都是复杂的、深奥的，需要极高的数学素养才能对这些概念有深刻的认识。那么，对于我们这些非数学专业、非硕士和博士的人来说，该如何掌握它们呢？依然把它们当成一个个组件就好。化繁为简——把复杂的问题归纳为相对固定和简单的套路，把复杂的模型整合为相对固定和简单的部件的组合。“把书读薄”就是这样一个概念——让学到的知识具有泛化性，让知识之间的共通性更为明显，从而降低认知新事物的难度。

在深度学习领域也有这样的套路吗？答案是肯定的。在深度神经网络的构建与训练中，有一个固定的套路，可以让我们在短时间内理解它的构建和训练方式。不管我们是自己搭建网络，还是阅读别人的代码，用这样的方式去做，思路都会清晰很多。

我们需要明确的对象包括：输入；输出；网络结构；损失函数。在一个深度学习项目中，只要这些对象都明确了，那么下面的问题就变得非常简单了。

6.6.1 输入

在本书的一开始我们就说过，机器学习的目的就是解决从 x 到 $y = f(x|\theta)$ 映射过程中 θ 的取值问题，其中网络的输入就是 x。

x 可以是一个数字、一个单词、一句话、一幅图片、一段声音……几乎可以是任何你期望的形式。只不过，x 必须以量化的形式输入。单词也好、图片也罢，都必须根据一定的规则进行编码，采用适当的方式进行处理，例如归一化（Normalization）、白化（Whitening）、灰度化、嵌入（Embedding）等，然后才能输入网络——总之，就是一些必要的预处理（Preprocessing）手段。随着我们介绍的工程案例数量的增加，输入的内容形式和处理技巧也会慢慢地具象化。

6.6.2 输出

一个输入的 x，经过网络中的 θ 的各种加工、计算，会产生一个输出值。我们先不管在网络

① 面向对象编程（Object Oriented Programming，OOP）是一种区别于传统的面向过程编程继承、多态、封装思想的程序架构方式。

中究竟是如何计算的，通常输出值都是整个网络模型希望输出的结果。

假设有一个深度神经网络，我们可以向其中输入一个用户的画像信息。这个画像信息的形式可能是

[性别 = 男，年龄 = 35，婚姻 = 已婚，房贷 = 300 万元……],

我们希望得到的输出是这个用户购买某种产品的概率。那么，在这个 Case 中，输出就是一个 0 到 1 之间的概率值，即一个表示概率的实数。

例如，我们希望有一个深度神经网络，向其中输入一幅图片，得到这幅图片表现某类物体的分类（如图 6-20 所示），那么，这个网络的输出就是一个独热向量（至少你希望是），或者一个表示其属于某个分类的概率为多少的多维向量。至于独热向量究竟“长”什么样子，后面我们会具体介绍。

图 6-20　图片分类

再如，我们希望有一个深度神经网络，向其中输入一段文字，能得到对应的语音。这就是一个比较典型的从文本到语音（Text to Speech，TTS）的声音合成项目的需求。这个网络输出的内容应该是一个语音文件或者一个梅尔倒谱 ①（期望在进行傅里叶反变换后通过还原得到波形文件）。

总之，输出的内容与具体需求直接相关。我们期望输出什么内容，就应该让模型在最后一层输出什么标签。深度学习中的标签（Label）并不是中文里的标签，而是一个术语，是指 $y = f(x|\theta)$ 中 y 的标准值（或者说期望值）。

6.6.3　网络结构

网络结构其实是很灵活的。简单地说，它就像一个管道，连接着输入和输出。在这个管道中，有各种根据我们所掌握的数学基础知识、工程经验设计出来的网络层（Layer）。常用的网络层有全连接层（Full Connection Layer）、卷积层（Convolutional Layer）、池化层（Pooling Layer）、循环神经层（Recurrent Neural Layer）等，会在后面详细介绍。

① Mel Frequency Cepstrum Coefficient（MFCC），一种音频频谱信息，在人类语音识别方面很有用。

每一层网络都相当于一个函数 $g(x|\theta)$。如图 6-21 所示，为了区分不同层之间的表达式和参数而使用了相应的下标。每一层都有一个输入，也就是上一层的输出；每一层都有一个输出，也就是下一层的输入；每一层都有自己的待定系数，用 θ 表示。样本从网络的“前面”输入，通过一层层 $g(x|\theta)$ 函数的加工，映射为其他的数值或者向量。

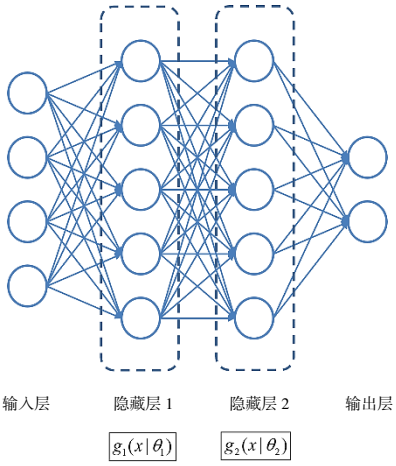


图 6-21 神经网络

只要网络的结构是确定的，那么映射的计算功能就是确定的。现在的问题是整个网络 $f(x|\theta)$ 的 θ 取什么值合适，具体到每一层中，就是 $g(x|\theta)$ 中的 θ 各取什么值合适。只要这些系数能够确定下来，这个网络就可以正常工作，就像一个函数的工作一样简单。如果想让网络正常工作，就需要进行训练，也就必须提到一个重要的概念——损失函数。

6.6.4 损失函数

损失函数（Loss Function），在一些资料中也称作代价函数（Cost Function），是机器学习训练过程中的一个非常重要的概念。损失函数通常是训练者自己定义的，它输出的值表示模型误差的大小。待定系数就是整个网络的待定系数 θ。

取什么样的 θ 值最为合理呢？如下两个条件是需要保证的。

- 条件一：取一个好的 θ，让 $f(x|\theta)$ 网络函数的工作过程可以通过一系列运算顺畅地把 x 加工成我们想要的输出值 y。误差要尽可能小，最好没有误差，这才是理想的。y 是我们设定的理想的输出值。例如，输入一幅老虎的图片，就应该输出一个描述为“老虎”的向量；输入一个数值，就应该输出一个和我们期望一致的运算结果。如果输出的与期望的不同，那就产生了误差，而这个误差应该能够被量化。

- 条件二：取一个好的 θ，在满足条件一的情况下，让 $f(x|\theta)$ 具有比较高的泛化性。也就是说，假如在训练时有人给了我 10000 个样本 x 及其对应的标签 y，让我训练 $f(x|\theta)$，在训练结束后，这 10000 个样本都能很好地通过 $f(x|\theta)$ 输出各自的 y 且误差在允许的范围内（当然，最好是零误差）。再给我一些同类的样本 x 和对应的 y，这些样本虽然没有参与训练，但训练好的 $f(x|\theta)$ 仍可以满足其映射关系。

只要满足这两个条件，$f(x|\theta)$ 就具有令人满意的低误差性和高泛化性了——完美。那么，如何描述误差比较好呢？对不同的场景来说，误差的描述肯定不尽相同，仅名称就有 MSE Loss、Cross Entropy Loss、CTC Loss 等。为了让大家能更容易地入门，不至于一上来就被弄到“头大”，我们只介绍 MSE Loss 和它的含义，其余的损失函数在碰到之后再介绍。

“MSE Loss”的全称是“Mean Squared Error Loss”，通常被翻译成“均方损失函数”。这种函数通常是用来解决回归问题的。也就是说，如果模型输出的是连续的实数值或者向量空间值，那么用 MSE Loss 来描述损失比较合适。该损失函数的表达式是这样的：

$$\text{Loss} = \frac{1}{2n}\sum_{i=1}^{n}(f(x_i|\theta) - y_i)^2.$$

又是一个看上去不太友好的公式，不过它的含义并不复杂。整个表达式的后半部分

$$\sum_{i=1}^{n}(f(x_i|\theta) - y_i)^2$$

表示：如果有 n 个样本，那么从第 1 个到第 n 个需要完成循环连加的工作；下标 i 表示序列。连加的内容是什么呢？每个样本要从网络的“入口”进入，通过网络的一系列计算输出一个值，用这个值减去标签值，然后计算其平方。对网络里面的 θ，只需要给定一个值，通过与其中某个样本 x_i 计算产生一个拟合值，将这个拟合值和期望的输出值 y_i 相减。毫无疑问：如果输出的拟合值和样本值 y_i 一样大，得到的结果就是 0；如果输出的值有偏差，就会产生一个或正或负的误差值。对于平方计算，0 的平方还是 0，而正负误差都变成了正数。显然，我们希望这个误差的平方尽可能小，最好是 0（这意味着模型精确）。前面的 $\frac{1}{2n}$ 也是有含义的，$\frac{1}{n}$ 表示在一个样本上的平均误差，而在分母上加一个 2 是希望在求导数的时候“削”掉它，看着好看一些而已……这么写有点平添麻烦的嫌疑，在具体的计算过程中其实不需要这么做。如果你在其他资料中看到

$$\text{Loss} = \frac{1}{n}\sum_{i=1}^{n}(f(x_i|\theta) - y_i)^2$$

这样的写法，也别觉得奇怪，事实上，这样的写法更符合均方损失函数的定义。

从表达式来看，均方损失函数反映了误差值的大小。那么，只要定义无误，我们应该考虑的就是通过什么方法让误差尽可能小——这就是训练（Training）过程。

6.6.5 求解极小值

损失函数明确以后，所谓训练的过程，就转化成一个求解极小值的过程了。该怎么理解呢？描述误差值大小的函数，可以这么表示：

$$\text{Loss} = \frac{1}{n}\sum_{i=1}^{n}(f(x_i|\theta) - y_i)^2.$$

在这个函数里面，谁是未知数、谁是已知数呢？x_i 和 y_i 是已知数，是样本；未知的是 θ。因此，我觉得这个损失函数这么写可能更为合适：

$$\text{Loss}(\theta) = \frac{1}{n}\sum_{i=1}^{n}(f(x_i|\theta) - y_i)^2.$$

这是一个以 θ 为自变量的函数，如果 θ 取某个值时能够保证整个损失函数的值为 0 或者足够小，应该就可以解决问题了。不用担心 $\text{Loss}(\theta)$ 会有小于 0 的情况，因为平方后求和的结果最小为 0。但是，θ 的取值不合适，就一定会导致误差。该怎么办？办法有的是，请往下看。

6.6.6 线性回归

为了更具象、更简洁地说明这个过程，我们以线性回归为例（一个只有一维输入和一维输出的不带激励函数的神经元）来讲解。

假设我们手里有 10 个点——10 组 x_i 及相应的 y_i，如表 6-1 所示。

表 6–1 样本

序　数	x_i	y_i
1	1	3.2
2	1.5	3.9
3	2	5.2
4	2.5	6.1
5	3	6.8
6	3.5	8.5
7	4	8.8
8	4.5	10
9	5	11.1
10	5.5	12.2

如果为这些样本画一幅图，大概如图 6-22 所示 ①。

① 这幅图是用 WPS 的表格功能画出来的。请大家支持国产软件，支持 WPS。

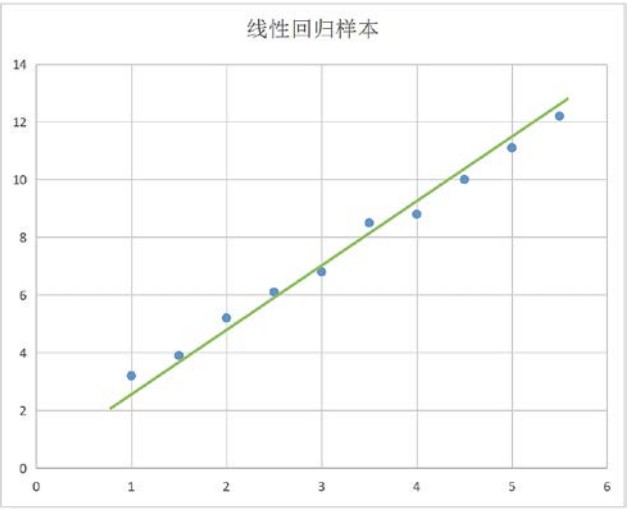


图 6-22　用 WPS 的表格功能绘制的线性回归样本示意图

当看到这样一幅图的时候，我立刻感觉，有一条线可以从这些点中间穿过去（就是中间这条线，是我后来画上去的），这条线就是非常典型的 $y = wx + b$ 函数的图象。这个单节点神经网络的输入和输出过程，如图 6-23 所示。

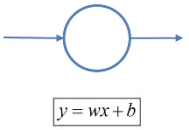


图 6-23　一维输入、一维输出的无激励函数神经元

当 10 个样本都通过这个单节点神经网络后，损失函数的表达式如下，其中 $n = 10$。

$$\text{Loss}(\theta) = \text{Loss}(w, b) = \sum_{i=1}^{n} (y_i - (wx_i + b))^2;$$

$$\text{Loss}(w, b) = \sum_{i=1}^{n} (x_i^2 w^2 + b^2 + 2x_i wb - 2y_i b - 2x_i y_i w + y_i^2).$$

在这里，我改变了损失函数的形式（不使用 $\frac{1}{n}$ 的形式），不仅使损失函数看上去简洁了一些，而且使极值点的位置保持不变（其实，乘以 $\frac{1}{n}$ 只影响了这条“抛物线”的开口大小，并不影响最低点的位置），参数 θ 也具体化成本示例中的 w 和 b 了。

将各样本产生的误差进行平方后，会得到 $\sum$ 里面的这个多项式，有 w^2 项、b^2 项、wb 项、w 项、b 项，以及后面的常数项 y_i^2。将 $\sum$ 完全展开，w^2 项、b^2 项、wb 项、w 项、b 项和常数项都会各自提取公因式并进行合并，变成下面这种形式。

$$\text{Loss} = Aw^2 + Bb^2 + Cwb + Dw + Eb + F.$$

其中，A、B、C、D、E、F 都是常数系数。

现在，我们得到了一个全局性的误差函数，其中的未知数是 w 和 b。我们要做的事情是，找到一个比较好的 w 和一个比较好的 b，使得整个 Loss 的值尽可能小，越接近 0 越好（说明拟合的误差越小）。我们虽然不知道这些值具体是什么，但可以把它的“近亲”——$z = f(x, y) = x^2 + y^2 + xy + x + y + 1$ 的图象画出来看看，如图 6-24 所示。

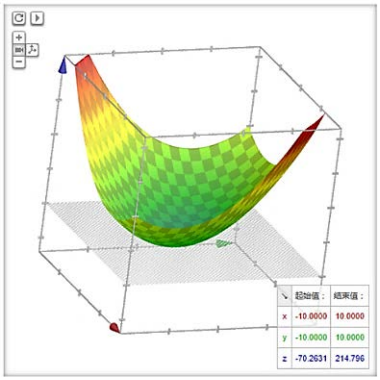


图 6-24 二维凸函数 $z = x^2 + y^2 + xy + x + y + 1$

该函数图象在 x、y、z 三维直角坐标系中看上去像一只碗。看到这个图象，我们的心就放下了一半：它确实和平面直角坐标系中的抛物线一样存在极值。所以，下面的问题就是求极值了。

解出“碗底”的坐标 (x, y)，也就是我们刚才说的 (w, b) 的取值点。这里的 (x, y) 相当于 (w, b)，尽管字母不同，但意义是相同的，且这个点就是保证全局误差 Loss 的解。

这个极值点容易解出来吗？按照在初中时学过的解一元二次方程 $y = ax^2 + bx + c$ 的套路，不管是极大值还是极小值，只要代入公式求解就行了。极值点是什么来着？

$$\left(\frac{-b}{2a}, \frac{4ac - b^2}{4a}\right),$$

即当 $x = \frac{-b}{2a}$ 的时候，$y = \frac{4ac-b^2}{4a}$。这就是极值点的坐标 (x, y)。

这种形式叫作解析解（Analytic Solution），是通过配方、移项等方法“拼凑”出来的。解的形式表示出来，仍然是一个标准的函数表达式，只不过自变量变成了 a、b、c。

现在的问题在于，大部分的方程是没有办法写出求根公式的，也没有办法用解析解表示其极值点的位置，至少目前从数学的角度看，这是无解的。然而，计算机本身也不会自动进行类似配方、移项这样的运算，它只能完成加、减、乘、除和比大小的运算。因此，只能用计算机本身的这些计算手段来实现了。这就是另一种方法，即迭代法（Iterative Method），也是用计算机求解一切方程的近似解的通用方法。例如，用刚刚介绍的这种方法求解损失函数

$$\text{Loss} = \sum_{i=1}^{n}(x_i^2 w^2 + b^2 + 2x_i wb - 2y_i b - 2x_i y_i w + y_i^2)$$

的极小值，该怎么操作呢？我们可以这样尝试一下。

首先，初始化一个 w_0 和一个 b_0。任何实数都可以，反正代入

$$\text{Loss} = \sum_{i=1}^{n}(y_i - (wx_i + b))^2$$

中的，都是可以输出一个 Loss_0 值的。在这个时候，$(w_0, b_0, \text{Loss}_0)$ 就会出现在“碗壁”的某个位置，而这个位置很可能离我们要找到的“碗底”很远。我们的前人也都知道，计算机有些时候不太“靠谱”，因为它自己没有任何“解题”思路，还得由我们告诉它怎样才能找到这个极值点。现在这个方法的应用已经很普遍了，梯度下降方法就是其中的一种（它也是用来解决凸优化问题的通用方法）。

如果这些字母让你迷茫，我们通过一个具象的例子来理解。$y = x^2 + 2$ 是一个典型的一元凸函数（如图 6-25 所示），刚才说的

$$z = f(x, y) = x^2 + y^2 + xy + x + y + 1$$

则是二元凸函数，三元及以上的高维凸函数也是有的，只是没办法画出图象而已。

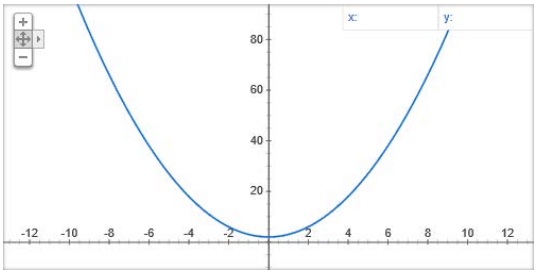


图 6-25　$f(x) = x^2 + 2$

对这种函数，要求其极小值，是有固定套路的，可以用一种被称作“挪挪看”的方法来求解。在 6.6.7 节中，我们就讨论一下这种一元二次函数的极值点是怎么找到的。

6.6.7　凸函数

假设我们不知道 $f(x) = x^2 + 2$ 的极小值在什么位置，也不想用解析解来表达这个位置，可以在函数曲线上随意取一个点

$$(x_0, x_0^2 + 2).$$

因为是随意取的点，所以十有八九不会恰好为极值点。假设我们开始取的是 (3,11) 这个点，而这

个点显然不是我们要求的极值点。没关系，在取了这个点之后，让计算机往它的两边“看”，看哪边低一些。例如，$x=2.8$ 和 $x=3.2$ 这两个值，分别对应的是

$$(2.8,9.84)$$

和

$$(3.2,12.24)$$

这两个点。我们很容易就能比较得出，$x=2.8$ 这边低一些——太好了，就往这边挪吧。下一次，就是比较 $x=2.6$ 和 $x=3.0$ 所对应的两个点了。$x=2.6$ 所对应的点的函数值比较小，就往这边挪。

按照这种方式，就可以在十几次比较以后，挪到极值点 $x=0$ 的位置了。看，即便不通过笔算解方程的方式，我们还是能用循环及加、减、乘、除这么“草根”的办法来求极值。

每次将 x 的值改变 0.2，只是为了给大家讲解这个示例而设置的，在现实中我们通常无法确定改变多少合适。一般来说，我们希望这种改变是自适应的，也就是说，该多挪的时候就多挪，该少挪的时候就少挪，挪到位了就别挪了——这才是理想的。如果步长太大，很容易一下子迈过“碗底”；如果步长太小，更新速度太慢。

梯度下降法就是为了解决这种问题而提出的。还用刚才这个例子来讨论：能不能让每次更新的值不都是 0.2，离极值点还远的时候就挪得“快”一些，离极值点比较近的时候就挪得“慢”一些，挪到位就不再挪动了？这个方法写出来是这么个形式：

$$x_{n+1}=x_n-\eta\frac{\mathrm{d}f(x)}{\mathrm{d}x}.$$

相信学过微积分的读者朋友对这样的公式会感觉很亲切。没学过的读者朋友也别着急，让我用“白话”解释一下它的含义是什么。

一个更新逻辑过程 x_{n+1} 和 x_n，分别表示两个临近迭代中的 x 的值，x_{n+1} 是 x_n 更新后的下一次迭代的值，每次更新时将 $x_n-\eta\frac{\mathrm{d}f(x)}{\mathrm{d}x}$ 的值赋给 x_{n+1}。其中，希腊字母 η 读作“伊塔”，称为“学习率”，即一个挪动步长的基数，所以也叫作“步长”。步长是在学习的开始，由编写程序的人赋值的。步长大就挪动得多，步长小就挪动得少。$\frac{\mathrm{d}f(x)}{\mathrm{d}x}$ 是 $f(x)$ 的导函数（或称导数），也可以记作 $f'(x)$。这个导数的概念，就是函数曲线上的切线斜率的概念。

在这里，我们用一点篇幅来补充介绍导数的概念。简单地说，导数就是“变化率”。例如，在物理学范畴，速度、位移和时间三者的关系是什么？按照我们在中学学习的物理学知识，通常这么表示：

$$s=vt.$$

速度通常表示为

$$v = \frac{s}{t},$$

就是位移相对于时间的变化率。对于开汽车这种场景来说，由于汽车通常不是匀速前进的，一会儿快，一会儿慢，v 也不是一个固定的值。但是没关系，速度 v 仍然可以表示成 $v(t)$，也就是一个随时间变化的函数。其具体定义是：

$$v(t) = \lim_{\Delta t \to 0} \frac{s(t + \Delta t) - s(t)}{\Delta t}.$$

这是在大学理工类专业高等数学课程的第一个学期就会学习的一个表达式，其中“lim”是“limit”的缩写，发音就是“limit”这个单词的英文发音，表示“极限”。具体表示什么的极限呢？就是 $s(t + \Delta t) - s(t)$ 和 Δt 比值的极限。也就是说，在某一时刻 t 到其后间隔 Δt 这段时间内产生的位移与 t 时刻的位移的差和这段时间的比值，而且希望这个时间差无限接近于 0。如果 $t = 0$，且 Δt 的取值对结果没有影响，那后面这个式子

$$\frac{s(t + \Delta t) - s(t)}{\Delta t}$$

就纯粹变成求平均速度了，和 $v = \frac{s}{t}$ 没有区别。但是，现在有了限制（就是刚刚说的这个限制），于是，在两个时间点的间隔无限趋近于 0 的过程中，位移的差距也无限趋近于 0。此时求其比值，得到的就是瞬时速度了，且时间轴上的每个点都是这样的。不用担心 Δt 无限趋近于 0 会导致“0 做除数”之类的不良后果，这是不会发生的，因为与这种情况同时发生的是分母上的 $s(t + \Delta t) - s(t)$ 无限趋近于 0。它们之间的倍数关系永远存在，就是瞬时速度。

在科学计算领域，或者说，在微积分领域，为了使研究和描述这类问题更加方便，定义了导函数（Derived Function）。其定义是

$$v(t_0) = \lim_{t \to t_0} \frac{s(t) - s(t_0)}{t - t_0}$$

或

$$v(t) = \lim_{\Delta \to t} \frac{s(t + \Delta) - s(t)}{\Delta},$$

表示用 s 对 t 求导数。$\mathrm{d}s$ 和 $\mathrm{d}t$ 分别指 Δs 和 Δt。$\frac{\mathrm{d}s}{\mathrm{d}t}$ 同样可以记作 $s'(t)$ 或者 s'，意为导数。

一个函数的导函数是一个新的函数，这个新函数上的每一个点的 y 值就是原函数的切线的斜率。如图 6-26 所示，对 $y = x^2$ 这个函数，要想知道函数曲线上每个点的斜率为多少，怎么办？用一条直线贴着 $y = x^2$ 的曲线“滚”一圈，记录每个时刻的切点 x 的值和切线的斜率 y，就得到了新的函数关系 $x \to y$。

关于导数，补充这么多就够了。

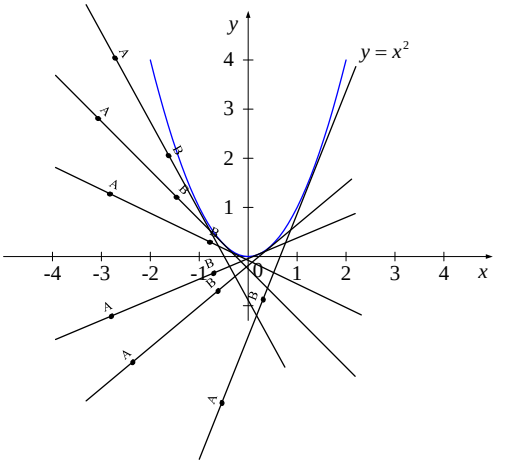


图 6-26　导函数的大小就是切线的斜率

我们回过头来看一下前面的例子。如图 6-27 所示，以函数 $f(x)=x^2+2$ 为例，$x=3$ 这一点的导数就是 (3,11) 这一点的斜率，因此，可以在函数的这一点做切线、求斜率。直接把 $f'(x)$ 的表达式求出来，即 $f'(x)=2x$[①]，然后把 $x=3$ 代入 $f'(x)$，得到相同的结果——$f(x)=x^2+2$ 中 (3,11) 这一点的斜率。

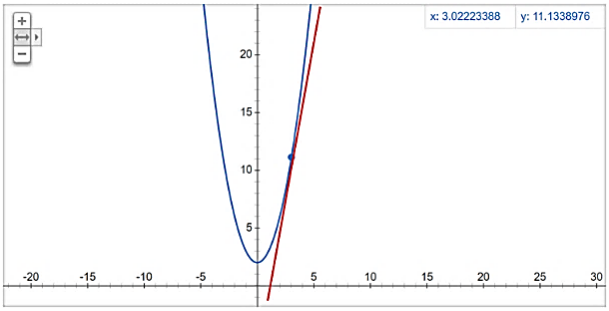


图 6-27　$f(x)=x^2+2$ 及其导函数 $f'(x)=2x$ 在 $x=3$ 处的状态

这一点的导数值看上去还是蛮大的，我替大家直接求解，得到 $f'(3)=6$。假如给 η 赋值 0.1，$-\eta\frac{\mathrm{d}f(x)}{\mathrm{d}x}$ 就应该等于 -0.6。因为 $x_n=3$，所以更新后 $x_{n+1}=2.4$。

如图 6-28 所示，进行下一次迭代，即 $x_n=2.4$ 时，$f'(2.4)=4.8$，那么 $-\eta\frac{\mathrm{d}f(x)}{\mathrm{d}x}=-0.48$，更新后 $x_{n+1}=1.92$。

① 高等数学的基础知识包括一些常用函数的导函数，可以直接查表得到，不用严格按照定义重新求解。

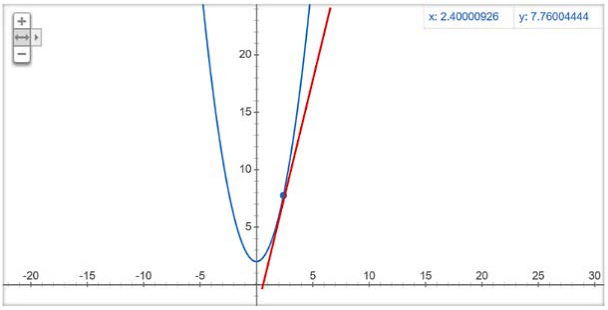


图 6-28　$f(x)=x^2+2$ 及其导函数 $f'(x)=2x$ 在 $x=2.4$ 处的状态

大家都看到了，移动的步长逐步减小，原因就是临近函数图象圆乎乎的底部时，由于斜率降低，更新量 $-\eta\frac{\mathrm{d}f(x)}{\mathrm{d}x}$ 的绝对值将会减小，更新时改变的量也会相应减小。

不知道细心的读者发现没有，更新原则

$$x_{n+1}=x_n-\eta\frac{\mathrm{d}f(x)}{\mathrm{d}x}$$

还有一个优势，就是没有向两边进行尝试，而是直接进行更新了。

这是为什么呢？就用函数 $f(x)=x^2+2$ 来说明吧。

- 当 $x_n=3$ 时，这个斜率是这一点的切线。由于斜率是一个正数，$-\eta\frac{\mathrm{d}f(x)}{\mathrm{d}x}$ 这一项一定是一个负数，更新后 x_{n+1} 会变小，往函数图象底部挪动。
- 当 $x_n=-3$ 时，这一点的斜率是一个负数，因此 $-\eta\frac{\mathrm{d}f(x)}{\mathrm{d}x}$ 这一项一定是一个正数，更新后 x_{n+1} 会变大，也会往函数图象底部挪动。

以上两点都是保证这种算法收敛的基本因素。这种图象底部圆乎乎的函数，就是凸函数。

从教科书上的定义来看，凸函数有着复杂和严格的定义：

$$f(ax+by)\leqslant af(x)+bf(y).$$

不过，我们可以记住它最简单、最基础的性质（也是一个必要条件）：在函数 $f(x)$ 上有任意两个变量 x_1 和 x_2，函数满足 $f(\frac{x_1+x_2}{2})\leqslant\frac{f(x_1)+f(x_2)}{2}$，如图 6-29 所示。

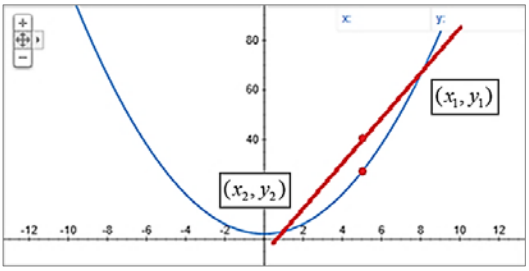


图 6-29　凸函数的特性

可以看到，这种函数的图象与 $f(x)=wx+b$ 的那种直勾勾的函数图象相比，具备了一种特性：要么是一条直线，即恰好满足 $f(\frac{x_1+x_2}{2})=\frac{f(x_1)+f(x_2)}{2}$ 时；要么是向一侧弯曲的曲线，而且是向“下方”弯曲的，即 $f(\frac{x_1+x_2}{2})<\frac{f(x_1)+f(x_2)}{2}$ 时。之所以叫作“凸”函数，其实是因为我们（观察者）从函数图象的下方看，函数图象向我们所在的方向“凸”出来了。

相信很多读者朋友可能会觉得这种说法太奇怪了，和自己的认知相反。如图 6-30 所示，从平视的角度看，这类函数图象就像一只碗或者一个盆，是“凹”进去的，可是人家不叫“凹”函数，非得反着来不可。其实嘛，习惯就好了。

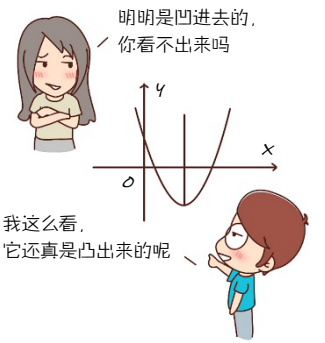


图 6-30　凹凸之争

如果最终我们把一个问题的求解过程，化简成一个在凸函数上求极值的过程，这个问题就可以解决了。在这种情况下，需要使用梯度下降法来求极值，核心思路就是在函数的曲线（曲面）上初始化一个点，然后沿着梯度下降的方向移动，直到函数值极点的位置。这个位置视具体的问题而定，可能是极小值，也可能是极大值（如果是凹函数，那就是梯度上升了）。很容易理解，是不是？

6.6.8　二元（多元）凸函数

别着急，我们快要摸清门道了。现在，对一维凸函数的极值，我们可以用编写程序的方法来求解了，那么对二维凸函数，该怎么办呢？我们来看函数

$$z=f(x,y)=x^2+8y^2,$$

其图象如图 6-31 所示。

在这个函数上，如果我们想从开始给定的一个点 (x_0,y_0)，通过迭代的方式挪到极值点，恐怕思路会跟之前不太一样。和一元凸函数不同，在这里最起码我们有四个方向可以尝试（一元凸函数只有两个方向）。不过，在更新方程

$$x_{n+1}=x_n-\eta\frac{\mathrm{d}f(x)}{\mathrm{d}x}$$

之中，我们发现了一个窍门：不用真的往两边进行尝试，这个更新方程本身就能在一个维度上“识别”这个方向。那么，在这两个维度上，应该有两个方程，分别是

$$x_{n+1} = x_n - \eta \frac{\partial f(x,y)}{\partial x}$$

和

$$y_{n+1} = y_n - \eta \frac{\partial f(x,y)}{\partial y}.$$

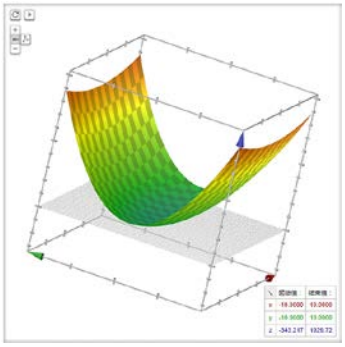


图 6-31　二元凸函数 $z = f(x,y) = x^2 + 8y^2$（1）

这里有一个新的符号出现了——∂。这个像反着写的数字 6 一样的符号，是数学中用来表示偏导数的符号。依然以函数 $z = f(x,y) = x^2 + 8y^2$ 为例，根据求导数的定义进行研究。$\frac{\partial f(x,y)}{\partial x}$ 读作“偏 f 偏 x”，含义是在 $z = f(x,y)$ 这个曲面上的点沿着平行于 x 轴的方向做切线，$\frac{\partial f(x,y)}{\partial x}$ 表示在点 (x,y) 上沿着平行于 x 轴方向所做的切线的斜率，如图 6-32 所示。

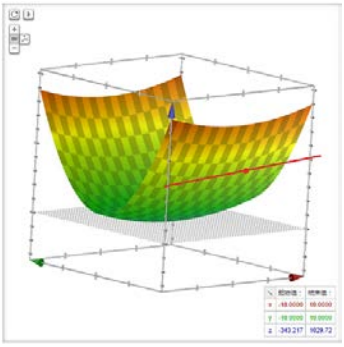


图 6-32　二元凸函数 $z = f(x,y) = x^2 + 8y^2$（2）

同理，$\frac{\partial f(x,y)}{\partial y}$ 的含义是在 $z = f(x,y)$ 这个曲面上的点沿着平行于 y 轴的方向做切线，$\frac{\partial f(x,y)}{\partial x}$ 表示在点 (x,y) 上沿着平行于 y 轴方向所做的切线的斜率，如图 6-33 所示。

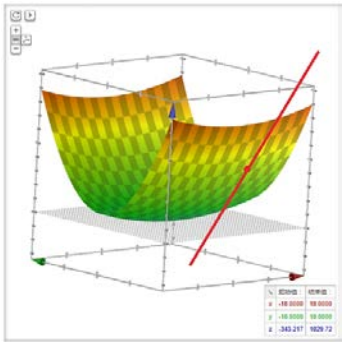


图 6-33 二元凸函数 $z=f(x,y)=x^2+8y^2$（3）

从数学的角度来说，$\frac{\partial f(x,y)}{\partial x}$ 和 $\frac{\partial f(x,y)}{\partial y}$ 的求法与 $\frac{\mathrm{d}f(x,y)}{\mathrm{d}x}$ 和 $\frac{\mathrm{d}f(x,y)}{\mathrm{d}y}$ 的求法一样。以 $\frac{\mathrm{d}f(x,y)}{\mathrm{d}x}$ 为例，就是在表达式 $z=f(x,y)$ 中，把 y 直接当成一个已知数（或者说系数）来看待，对 x 求导数，$\frac{\mathrm{d}f(x,y)}{\mathrm{d}y}$ 同理。在这里，$\frac{\mathrm{d}f(x,y)}{\mathrm{d}x}$ 和 $\frac{\mathrm{d}f(x,y)}{\mathrm{d}y}$ 这种写法在数学层面就显得不够专业了，因为这种写法只是为了表达这个过程与一元函数求导数没有差别。通过求导，可以分别得到在 $z=f(x,y)=x^2+8y^2$ 这个二元函数中，

$$\frac{\partial f(x,y)}{\partial x}=2x,$$

$$\frac{\mathrm{d}f(x,y)}{\mathrm{d}y}=16y.$$

那么，对任何一个点 (x,y)，我们都知道其 $\frac{\partial f(x,y)}{\partial x}$ 和 $\frac{\partial f(x,y)}{\partial y}$ 分别是多少了。例如，(3,4) 这个点的 $\frac{\partial f(x,y)}{\partial x}$ 和 $\frac{\partial f(x,y)}{\partial y}$ 为 (6,64)。这时，更新方程就可以开始工作了：

$$x_{n+1}=x_n-\eta\frac{\partial f(x,y)}{\partial x};$$

$$y_{n+1}=y_n-\eta\frac{\partial f(x,y)}{\partial y}.$$

不过，对于计算机有限的运算能力来说，计算资源永远都是不足的。在实际的深度学习网络中，极可能有几千万甚至上亿个维度、几十亿个维度需要更新！所以，从效率层面考虑，也希望这种更新能够产生最好的收敛效果。所谓“收敛”，就是通过多次迭代，逐步逼近想要求的值的过程。显然，在准确度相当的情况下，收敛速度更快的方法会更受欢迎。在梯度下降法中有多维度的选择。例如，$z=f(x,y)$ 就有两个维度。哪一种更新原则的收敛速度最快呢，有方法可循吗？当然有。

如图 6-34 所示，在一个三维空间中，从“山顶”往“山脚”走，用时最短的路径是沿着最陡峭的方向的那一条。这里有一个名词，叫作“梯度”，记作

$$\nabla = \left(\frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y}\right),$$

在形式上是两个方向上的偏导数。在每次更新时，就是用 η 分别乘以两个方向上的偏导数，让它们各自完成更新量：

$$x_{n+1} = x_n - \eta \frac{\partial f(x,y)}{\partial x};$$

$$y_{n+1} = y_n - \eta \frac{\partial f(x,y)}{\partial y}.$$

图 6-34　梯度下降

如果变量很多，例如不只有 x 和 y 两个，而是有 1000 个 w，该怎么办？也是一样的，把它们表示成

$$\nabla = \left(\frac{\partial f(w)}{\partial w_1}, \frac{\partial f(w)}{\partial w_2}, \ldots, \frac{\partial f(w)}{\partial w_{1000}}\right),$$

再分别乘以 η，得到

$$w^i_{n+1} = {w^i}_n - \eta \frac{\partial f(w)}{\partial w^i}.$$

w 的上标 i 表示第 i 个 w，w 的下标 n 和 $n+1$ 表示迭代的次数。

到这里，我们基本理解了梯度下降法（最速梯度下降法）在多元凸函数上更新时的步骤和原理。那么，如何解决训练问题呢？我想你心里已经明白了八九分了。只要能够把残差函数 Loss 描述成由待定的若干个 w 所描述的凸函数 $\mathrm{Loss}(w)$，就可以使用梯度下降法，即用最快的方法更新 w 的各个维度。最后，找到满足 $\mathrm{Loss}(w)$ 的极值点，就算大功告成了。

6.6.9　导数补充

关于导数，我们讨论一下由多个函数组合形成的新函数的导数应该怎么求的问题。

我们通常会遇到函数的其他形式或者变种，例如

$$y = g(x)f(x),$$

我们权且称之为“连乘型”。

还有一种形式

$$y = g\big(f(x)\big),$$

我们称之为“嵌套型”。

基本上，这两种方式的组合可以用来表示绝大多数复杂函数的结合过程。也就是说，如果我们了解了这两种函数的求导过程，就相当于了解了绝大多数复杂函数的求导过程。

1. 连乘型

我们先来看看“连乘型”的求导过程。

根据导数的定义，我们最终要求的是 $\frac{\mathrm{d}y}{\mathrm{d}x}$，因此要用标准的数学极限推导方法试求一下。

$$\begin{aligned}
\frac{\mathrm{d}y}{\mathrm{d}x} &= \frac{\mathrm{d}[g(x)f(x)]}{\mathrm{d}x} \\
&= \lim_{\Delta\to 0}\frac{g(x+\Delta)f(x+\Delta)-g(x)f(x)}{\Delta} \\
&= \lim_{\Delta\to 0}\frac{g(x+\Delta)f(x+\Delta)-g(x)f(x)-g(x)f(x+\Delta)\Delta+g(x)f(x+\Delta)\Delta}{\Delta} \\
&= \lim_{\Delta\to 0}[\frac{g(x+\Delta)-g(x)}{\Delta}f(x+\Delta)+g(x)\frac{f(x+\Delta)-f(x)}{\Delta}] \\
&= \lim_{\Delta\to 0}\frac{g(x+\Delta)-g(x)}{\Delta}f(x+\Delta)+\lim_{\Delta\to 0}g(x)\frac{f(x+\Delta)-f(x)}{\Delta} \\
&= g'(x)f(x)+g(x)f'(x).
\end{aligned}$$

“连乘型”的 $[g(x)f(x)]' = g'(x)f(x)+g(x)f'(x)$，很简单吧？

如果看着这些推导过程觉得头晕眼花，也没关系，听一下“白话”的解释。对于“连乘型”的函数组合，新函数的斜率相当于两部的分线性叠加（求和），一部分是 $f(x)$ 与 $g(x)$ 的斜率相乘，另一部分是 $g(x)$ 与 $f(x)$ 的斜率相乘。

2. 嵌套型

我们再看看“嵌套型”的求导怎么做。

对于 $y = g(f(x))$，我们需要求 $[g(f(x))]'$。为了降低难度，设 $f(x) = z$。实际上，$y = g(f(x))$ 被代换成了

$$y = g(z),$$
$$z = f(x)$$

两个等式。

如果 $y=g(f(x))$ 在 x 上存在导数，那么

$$\lim_{\Delta z\to 0}\frac{\Delta y}{\Delta z}=g'(z)$$

中的 z 是我们刚刚请来帮忙的，我们把它看成一个独立的变量就可以了。

在 $\frac{\Delta y}{\Delta z}=g'(z)+\alpha$ 中，α 是一个无穷小的数，我们可以认为它极其接近 0。在等式两边同时乘以 Δz，会得到

$$\Delta y=g'(z)\Delta z+\alpha\Delta z.$$

在等式两边同时除以 Δx，会得到

$$\frac{\Delta y}{\Delta x}=g'(z)\frac{\Delta z}{\Delta x}+\alpha\frac{\Delta z}{\Delta x}.$$

由于 $\lim\limits_{\Delta x\to 0}\alpha=0$，化简得到

$$\lim_{\Delta x\to 0}\frac{\Delta y}{\Delta x}=g'(z)f'(x).$$

也就是说，$y=g(f(x))$ 这种“嵌套型”的求导，其实就是把函数一层层包起来，用变量代换后，将一层层求导的导数相乘。

那么，扩展一下，下面这个表达式也就成立了：

$$\Big[u\Big(v\Big(g\big(f(x)\big)\Big)\Big)\Big]'=\frac{\mathrm{d}u}{\mathrm{d}v}\frac{\mathrm{d}v}{\mathrm{d}g}\frac{\mathrm{d}g}{\mathrm{d}f}\frac{\mathrm{d}f}{\mathrm{d}x}=u'(v)v'(g)g'(f)f'(x).$$

形式看着虽然乱，但是非常容易理解。如果一个函数在一个点 $f(x)$ 的变化率为 3 倍，而在同一个点 $g(x)$ 的变化率为 5 倍，那么，将它们嵌套在一起，就等于进行了两个变化率的叠加——15 倍。如果感觉这个推导过程太“烧脑”，那就别深究了，直接记住结论就好。实际上，在编写程序的过程中，也不会直接这么使用。

6.6.10　导数怎么求

问题来了：导数怎么求？这一节的内容很重要。我们先来了解一下，如何通过一段程序实现用梯度下降法更新 $\mathrm{Loss}(w)$ 这一过程。

在不太“靠谱”的计算机用加、减、乘、除进行更新的过程中，有一个问题我们在前面没有提到——怎么求导数？学过高等数学或数学分析的读者朋友也别高兴得太早，损失函数 $\mathrm{Loss}(w)$ 一般会呈现一种看上去非常“不规则”的样子，没办法通过查表得到导数的表达式，因此，更不可能通过代入当时这一点的 $\boldsymbol{x}$ 向量（n 个维度）来求出各个方向上的偏导数的值了。

不过，我们仍然有办法求出一个偏导数在某个点的大概值，并通过这个值进行判断。回忆一

下偏导数的几何定义——切线斜率。我们能不能用别的方法求出切线斜率呢？

还是以一元二次函数 $f(x)=x^2+2$ 为例，如图 6-35 所示。给出一个初始化的点 $(4,18)$，当 $x_{n+1}=x_n-\eta\frac{\mathrm{d}f(x)}{\mathrm{d}x}$ 时，$\frac{\mathrm{d}f(x)}{x}$ 的值应该是多少呢？

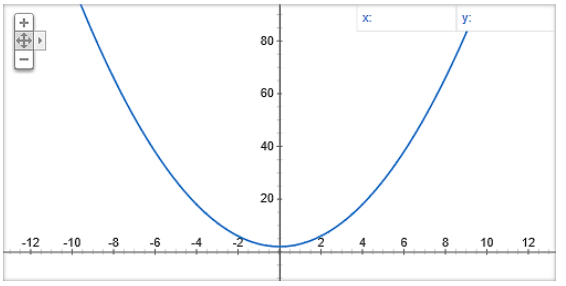


图 6-35　$f(x)=x^2+2$

既然是求切线，就要看看切线的定义：

$$\lim_{\Delta\to 0}\frac{f(x+\Delta)-f(x)}{\Delta}.$$

其实，这也是 $\frac{\mathrm{d}f(x)}{x}$ 的定义。

分子是 $f(x+\Delta)-f(x)$，也就是在任何一个 x 的取值上，向其旁边挪动 Δ 的位置。在数学和物理领域，Δ 专门用来表示一个很小的差值（注意：这里没有定义 Δ 的正负值）。所以，分子 $f(x+\Delta)-f(x)$ 的含义就是，将 x 向旁边挪动 Δ 的位置后，再看 $f(x+\Delta)$ 与当前 $f(x)$ 的差值。我们记住它的含义。

分母是一个 Δ。这个 Δ 是一个很小的差值，没有太多好说的。

再来看整个表达式。$\lim\limits_{\Delta\to 0}\frac{f(x+\Delta)-f(x)}{\Delta}$ 的含义就是，让这个比值中的 Δ 无限趋近于 0，但不能等于 0（0 做除数的情况在高等数学中还没有相应的解释）。

如图 6-35 所示，在 Δ 无限趋近于 0 时，不论它是一个正数还是一个负数，最后都会趋近于一个值，即这个点的斜率。$f(x+\Delta)-f(x)$ 是三角形直边的高，Δ 是三角形底边的长。

如果情况真的是这样，我们就可以用这样的表达式来近似代替导函数的斜率了。还是以点 $(4,18)$ 为例，对这一点的斜率，我们用 $\frac{f(x+0.001)-f(x)}{0.001}$ 来试试看。

$$\frac{f(x+0.001)-f(x)}{0.001}=\frac{18.008001-18}{0.001}=8.001\approx 8.$$

得到的值相对比较精确。

既然有了这种方法，那么在任何一个维度上，只要函数是可导[①]的，我们都能求出 $-\eta\frac{\mathrm{d}f(x)}{\mathrm{d}x}$

① 并非所有的函数都有导数。并非所有的函数在每个点上都有导数的定义。在这里就不进行更多的讨论了。

的近似值，并更新 $x_{n+1}=x_n-\eta\frac{\mathrm{d}f(x)}{\mathrm{d}x}$。不用担心 $8.001\approx 8$ 这种误差会产生叠加效果，最终导致求解出现问题，因为它在应用的过程中仅用来确定一个更新的量。在更新的过程中，就是用精确的方式更新了 n 维向量 $\boldsymbol{x}$ 的某个维度的值而已。在本节的示例中，不过是步长与理想的 $\eta\frac{\mathrm{d}f(x)}{\mathrm{d}x}$ 差了 0.001η 而已，我们完全不必为此而纠结。

6.6.11　“串联”的神经元

完成前面介绍的工作基本之后，我们就可以直面两个神经元“串联”的情况了，如图 6-36 所示。要知道，在深度学习中，神经元的“串联”才是常态。

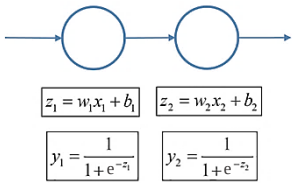


图 6-36　“串联”的神经元

如果要对两个前后相接的神经元进行训练，该怎么办呢？一个神经元怎么训练，两个神经元就怎么训练。在入口位置放 x，在出口位置放 y，然后初始化 θ，让每个样本都能通过网络的映射产生拟合的结果 $f(x|\theta)$。接下来，构造损失函数

$$\mathrm{Loss}=\frac{1}{n}\sum_{i=1}^{n}(f(x_i|\theta)-y_i)^2,$$

实现梯度下降，就可以了。不论将 w_1、b_1、w_2、b_2 初始化为什么值，损失函数 Loss 的值都恒为非负数。

现在，我们开始“挪动”w_1、b_1、w_2、b_2 这四个待定系数，使 Loss 的值逐步减小。

似曾相识的过程又出现了，需要通过如下四个表达式进行更新：

$$(w_1)^n=(w_1)^{n-1}-\eta\frac{\partial\mathrm{Loss}}{\partial w_1};$$

$$(b_1)^n=(b_1)^{n-1}-\eta\frac{\partial\mathrm{Loss}}{\partial b_1};$$

$$(w_2)^n=(w_2)^{n-1}-\eta\frac{\partial\mathrm{Loss}}{\partial w_2};$$

$$(b_2)^n=(b_2)^{n-1}-\eta\frac{\partial\mathrm{Loss}}{\partial b_2}.$$

大部分内容都很容易理解，无非是在一只五维空间里的“碗”中找“碗底”的过程。问题在于，$\frac{\partial\mathrm{Loss}}{\partial w_1}$、$\frac{\partial\mathrm{Loss}}{\partial b_1}$、$\frac{\partial\mathrm{Loss}}{\partial w_2}$、$\frac{\partial\mathrm{Loss}}{\partial b_2}$ 这四个值该怎么求。如果求不出来，就要面临在凸优化梯度下降过

程中因挪动过大而越过“碗底”的问题。

办法还是有的。在 6.6.9 节中我们学习了两种特殊的导函数求解方法，一种是“连乘型”，另一种是“嵌套型”，现在算是派上用场了。

首先，我们使用一个技巧，对整个函数进行变形：

$$\text{Loss}=\frac{1}{n}\sum_{i=1}^{n}(f(x_i|\theta)-y_i)^2 \Rightarrow \text{Loss}(w_1,b_1,w_2,b_2)=\frac{1}{2n}\sum_{i=1}^{n}(f(x_i|w_1,b_1,w_2,b_2)-y_i)^2.$$

上式也可以简写成

$$\text{Loss}=\frac{1}{2}\sum_{i=1}^{n}(f(x_i)-y_i)^2.$$

在函数前面配上 $\frac{1}{2}$，主要是为了一会儿“削”起来方便。

然后，用“嵌套型”函数的求导特点来求导。“嵌套型”的学名叫作“链式法则”（Chain Rule），先求离输出端最近的 Loss 对 w_2、b_2 的偏导数。

$$\begin{aligned}\frac{\partial \text{Loss}}{\partial b_2}&=\frac{\partial \frac{1}{2}\sum_{i=1}^{n}(f(x_i)-y_i)^2}{\partial b_2}\\&=[\sum_{i=1}^{n}(f(x_i)-y_i)]\frac{\partial f(x_i)}{\partial b_2}\\&=[\sum_{i=1}^{n}(y_2-y_i)]\frac{\partial y_2}{\partial z_2}\frac{\partial z_2}{\partial b_2};\end{aligned}$$

$$\begin{aligned}\frac{\partial \text{Loss}}{\partial w_2}&=\frac{\partial \frac{1}{2}\sum_{i=1}^{n}(f(x_i)-y_i)^2}{\partial w_2}\\&=[\sum_{i=1}^{n}(f(x_i)-y_i)]\frac{\partial f(x_i)}{\partial w_2}\\&=[\sum_{i=1}^{n}(y_2-y_i)]\frac{\partial y_2}{\partial z_2}\frac{\partial z_2}{\partial w_2}.\end{aligned}$$

以上就是后面这个神经元的两个待定系数 b_2 和 w_2 的表达式。由于 $\frac{\partial z_2}{\partial b_2}=1$，

$$\frac{\partial \text{Loss}}{\partial b_2}=[\sum_{i=1}^{n}(y_2-y_i)]\frac{\partial y_2}{\partial z_2}$$

就是求解“路径”最短的表达式，其次是

$$\frac{\partial \text{Loss}}{\partial w_2}=[\sum_{i=1}^{n}(y_2-y_i)]\frac{\partial y_2}{\partial z_2}\frac{\partial z_2}{\partial w_2}.$$

同理，继续推导，就是利用链式法则求对 b_1 和 w_1 的导数了。

$$\begin{aligned}\frac{\partial \text{Loss}}{\partial b_1} &= \frac{\partial \frac{1}{2}\sum_{i=1}^{n}(y_2-y_i)^2}{\partial b_1} \\ &= [\sum_{i=1}^{n}(y_2-y_i)]\frac{\partial y_2}{\partial b_1} \\ &= [\sum_{i=1}^{n}(y_2-y_i)]\frac{\partial y_2}{\partial z_2}\frac{\partial z_2}{\partial y_1}\frac{\partial y_1}{\partial z_1}\frac{\partial y_1}{\partial b_1};\end{aligned}$$

$$\begin{aligned}\frac{\partial \text{Loss}}{\partial w_1} &= \frac{\partial \frac{1}{2}\sum_{i=1}^{n}(y_2-y_i)^2}{\partial w_1} \\ &= [\sum_{i=1}^{n}(y_2-y_i)]\frac{\partial y_2}{\partial w_1} \\ &= [\sum_{i=1}^{n}(y_2-y_i)]\frac{\partial y_2}{\partial z_2}\frac{\partial z_2}{\partial y_1}\frac{\partial y_1}{\partial z_1}\frac{\partial y_1}{\partial w_1}.\end{aligned}$$

现在，我们就可以把每次更新的四个表达式的具体值求出来了。

$$(w_h)^n = (w_h)^{n-1} - \eta\frac{\partial \text{Loss}}{\partial w_h};$$

$$(b_h)^n = (b_h)^{n-1} - \eta\frac{\partial \text{Loss}}{\partial b_h};$$

$$(w_o)^n = (w_o)^{n-1} - \eta\frac{\partial \text{Loss}}{\partial w_o};$$

$$(b_o)^n = (b_o)^{n-1} - \eta\frac{\partial \text{Loss}}{\partial b_o}.$$

接下来，我们只要按照表达式进行更新就可以了。在其他结构的网络中，可能会有如下小小的区别。

- 区别一：在训练样本很多的情况下，Σ运算的次数会多一些。如果有 1000 个样本，就要进行 1000 次这样的运算；如果有 10000 个样本，就要进行 10000 次这样的运算。
- 区别二：如果网络层数很多，偏导数连乘的项就会很长。在本节中，2 层网络就出现了 4 个连乘项，如果网络有 10 层，就会有 20 个连乘项。
- 区别三：如果网络非常复杂，例如在一层网络中不止有 1 个节点，那么在某个节点上，系数的偏导数会通过多条路径传播。因为在这个节点的后面，会有不止 1 个节点把它的输出当成自己的输入，从而形成多个“嵌套型”关系。

6.6.12　模型的工作

当训练结束后，模型就可以开始工作了。

模型的工作过程是非常简单的，就是把一个输入的 x 代入训练好的 $y = f(x|\theta)$ 中去，使它输出一个 y。这个过程所消耗的时间比训练过程要短得多，毕竟训练过程需要通过迭代法的一次次计算来逼近要求的解，而模型的工作过程就是一轮普通的加、减、乘、除运算。

可以说，几乎所有的机器学习算法模型都会体现出这样的特点：训练的时间很长，而一次工作的时间很短。“训练”是一个误差反向传播过程，“工作”是一个样本正向传播过程。

6.6.13 理解损失函数

损失函数是机器学习模型中至关重要的因素。有很多靠谱的方法可以用来求解一个函数的极大值或极小值，对它的优化过程也已经比较成熟了。而针对“最优化问题”，仍然有很多数学家不断提出新的求解方法，期望以更高的效率、更精确的手段找到函数的极大值或极小值。

通前面的两个例子我们可以知道，损失函数的主要目的是描述误差的大小——这样理解是没有问题的。不过，在后面的例子中，我们会不断看到一个现象：损失函数可能会比较复杂，甚至可能由多个分项组成。这些分项通常以求和或者加权求和的形式组成一个完整的损失函数，写出来大概是这样的：

$$\text{Loss} = \alpha_1 \text{Loss}_1 + \alpha_2 \text{Loss}_2 + \ldots + \alpha_3 \text{Loss}_3.$$

其中，每个分项都有自己的意义和目的，前面都配有一个比例 α。简单地说，采用这种形式的目的是使这些分项的损失值尽可能小，尤其是 α 的值比较大的那些项目，在完整的损失评价中对它更为敏感（或者重视）。

很多项目之所以需要在损失函数中配置多个项目，其实就是为了增加对优化的限制条件（或者说，对模型训练的限制条件）。增加的项目越多，说明训练中的优化限制条件越多，因为这表示我们关心这个值对模型的影响。根据我在工程方面的经验，限制条件设置得越合理、数量越多，就越容易对模型正确收敛产生正向影响。这也是帮助“无脑”的机器人“思考”的手段之一。

到目前为止，我们在深度学习领域落地应用中看到的比较好的模型，要么是对输入样本的人为降维，要么是根据某些理论或研究成果构造特殊的损失函数、增加收敛的限制条件，使得收敛更为稳定，大概率向着实验者的目标进行优化。后者是深度学习界更为推崇的，大家在学习中不妨多加注意，看看情况是不是这样。

6.7 深度学习的优势

神经网络，尤其是深度神经网络，之所以这么吸引人，主要原因是它能够通过大量的线性分类器和非线性关系的组合来解决平时让我们感到棘手的线性不可分问题。

6.7.1　线性和非线性的叠加

一个深度神经网络包含大量的线性和非线性组件，并提供大量的待定系数。线性组件就是每个神经元前半部分的 $y = \boldsymbol{wx} + b$，非线性组件的就是后面的非线性激励函数。在 6.6 节中，我们介绍了 Sigmoid、Tanh、ReLU 三种常见的激励函数。那么，非线性函数的优势是什么呢？下面我们用 Sigmoid 函数来拟合一个布尔运算的关系。一个与或运算过程，如图 6-37 所示。

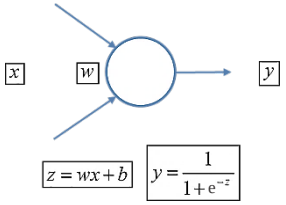


图 6-37　与或运算

这样一个包含两个输入和一个输出的线性单元，可以拟合出一个与运算吗？如果我们手动配置参数，令 $w_1 = 20$、$w_2 = 20$ 且 $b = -30$，就会有如表 6-2 所示的输出。

表 6–2　与运算

x_1	x_2	y 的期望值	y 的真实值
1	1	1	0.999954602
1	0	0	4.53979E-05
0	1	0	4.53979E-05
0	0	0	9.35762E-14

在“y 的真实值”一列，后面的三个数值使用了科学计数法，“E-05”的含义是“乘以 10^{-5}”（一个非常接近 0 的数字）。也就是说，一个有两个维度输入的神经元，可以在误差极小的情况下完成一个与运算的拟合。

或运算在输入的格式上和与运算没有区别，只不过需要让 $w_1 = 20$、$w_2 = 20$ 且 $b = -10$。其输出如表 6-3 所示。

表 6–3　或运算

x_1	x_2	y 的期望值	y 的真实值
1	1	1	1
1	0	1	0.999954602
0	1	1	0.999954602
0	0	0	4.53979E-05

可以看出，仍然在误差极小的情况下满足了或运算的输出需求。

非运算略显特殊，因为它是“一进一出”形式的，如图 6-38 所示。

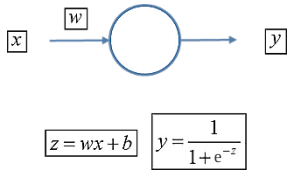


图 6-38　非运算

非运算要求输出的内容比较简单，也同样可以被满足。例如，让 $w=-20$ 且 $b=10$，如表 6-4 所示。

表 6–4　非运算

x	y 的期望值	y 的真实值
1	0	4.53979E-05
0	1	0.999954602

可以看出，仍然可以获得一个质量很高的输出过程。

通过本节前面的例子我们可以明显地看出，使用 Sigmoid 这样的函数就可以完成与、或、非运算。在这里，参数是直接人为指定的。而在真实的训练中，参数是通过定义损失函数，然后把这些 w 和 b“挪来挪去”得到（学到）的。

这就是非线性激励函数的能力。

对于 ReLU 这样的函数，我们可以发挥一下想象力。它的非线性形式是什么样的呢？一个值通过 $y=wx+b$ 进行映射后，就可以使用 ReLU 函数，保留大于 0 的部分，而对小于 0 的部分，不管值是多少，都取 0。如果一幅图像上的每一个点都被 ReLU 函数处理过，形成的新图像是什么样子的呢？对小于 0 的部分都取 0，即都可以“感性”地理解为“黑色”；对大于 0 的部分则做了保留，产生了“明亮”程度不同的类似灰度图的图像（如图 6-39 所示）。

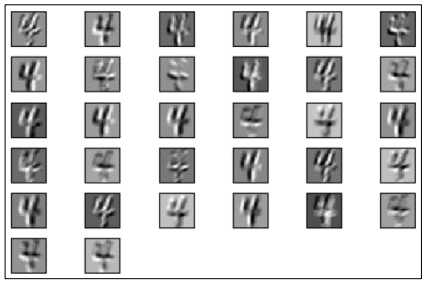


图 6-39　ReLU 函数处理后形成的可视化特征图

这就好像滤波器：“无视”一些期望边界以外的值，把它们直接变成“黑色”的；对“明亮”的部分，以量化的方式进行进一步的处理（后面的层对这些“明亮”的部分敏感）。例如，在一个完整的卷积神经网络中，要想对一辆汽车的图片进行分类，就要让它在特征图（Feature Map）上通过卷积产生不同的刺激值，将这些刺激值进行可视化处理，得到深浅不一的光斑（如图 6-40 所示）。如何才能让提取功能正常、有效地工作呢？那就要靠卷积核中 $y = wx + b$ 部分的筛选机制来实现了。训练的过程，就是训练这些 w 和 b，让它们变成满足损失函数约束条件的解。

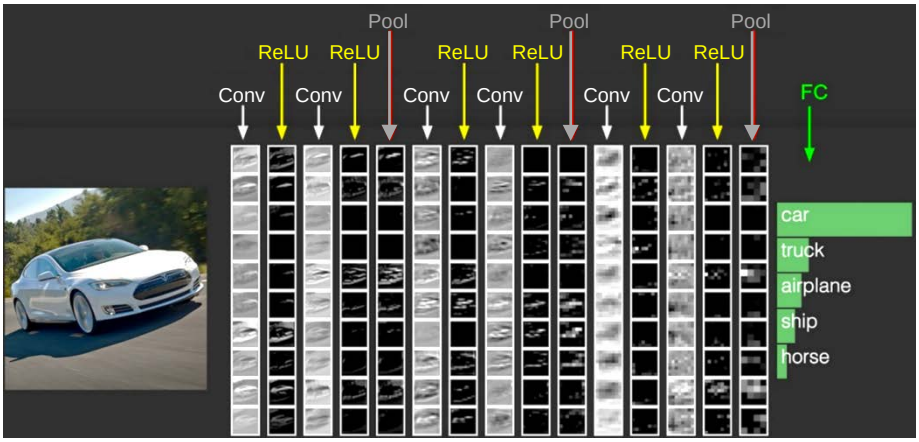


图 6-40　可视化特征图

6.7.2　不用再提取特征

有了非线性的拟合能力之后，深度神经网络的能力就有了比较显著的不同。

在以往我们使用的朴素贝叶斯、决策树、支持向量机等传统机器学习分类器模型中，提取特征是一项非常重要的前置工作，也就是说，人类在驱使这些分类器进行训练之前，要把大量的样本数据整理出来，“干干净净”地提取其中能够清晰量化的数据维度。否则，这些基于概率和空间距离的线性分类器是没办法正常工作的。

然而，由于数量巨大的线性分类器的堆叠和卷积神经网络的使用，以及对噪声的忍耐能力，对于从多通道数据投射出来的不同特征偏向的敏感程度，神经网络会自动选择重视或者无视。这样，在处理的时候，对人类需要掌握的技巧就没有那么高的要求了（这就是通常所说的 End-to-End 的训练方式）。不需要由人来完成数据特征的提取，也是深度学习的门槛不算太高的原因之一。于是，在处理某个行业内的应用时，即使没有丰富的行业经验和基础知识，也可以通过神经网络来提取数据的特征。

以前，如果想完成图片分类任务，哪怕实现一个非常简单且功能有限的 OCR[①]应用，都需要

① Optical Character Recognition，译为“光学字符识别”，是一种将图像中的文字识别出来的技术。

大量的计算机图形学知识，事先由人进行特征提取、分辨特征的有效性并反复进行实验。而现在，如果我们想让一个机器人模型完成图片识别工作，需要做的事情就是对图片进行有限且套路清晰的预处理，然后直接使用大量的样本进行堆叠。这样，深度学习网络就可以通过损失函数的约束，让网络中数以百万的 w 和 b 变成满足我们需求的样子了。

这才是基于深度学习的人工智能吸引人的地方。它不仅在一定程度上降低了自动化应用的门槛，更降低了人力成本。客观地说，就是将不确定性居多的人力成本转化成更为确定的硬件资源成本。

6.7.3　处理线性不可分

了解了前面介绍的特性，神经网络最神奇的地方就比较容易理解了——由大量的线性分类器模拟的堆叠，使得线性不可分的问题变得可描述了。

SVM[①]也拥有一定的能力来处理线性不可分的问题，但它是利用维度的引入（或者说升维）来解决这个问题的。而神经网络的每一个神经元都是一个线性分类器，所以，神经网络能且只能通过线性分类器的组合来解决线性不可分的问题。下面，我们通过一个例子来了解一下。

如图 6-41 所示，在一个二维空间中有一个不规则的四边形，我们想用一条线（线性分类器）把它分开，并保证其一侧是这个四边形内所有的点（称为“1 类”），另一侧是其他的点（称为“0 类”）。

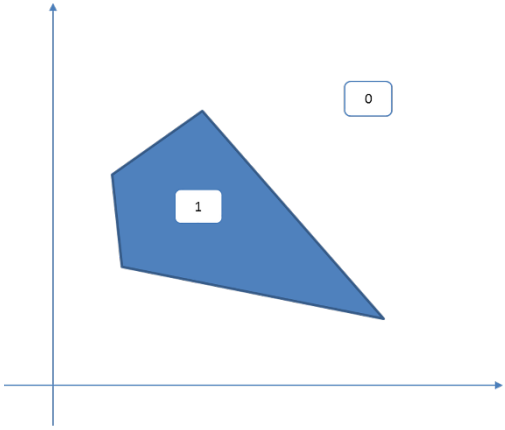


图 6-41　不规则的四边形分类

这简直是不可能实现的，因为不管怎么画线，都会使得其中至少有一个类“不纯”。没关系，我们大不了画 4 条线，如图 6-42 所示。

① Support Vector Machine，译为“支持向量机”，是在传统机器学习中使用的一种分类和回归算法。

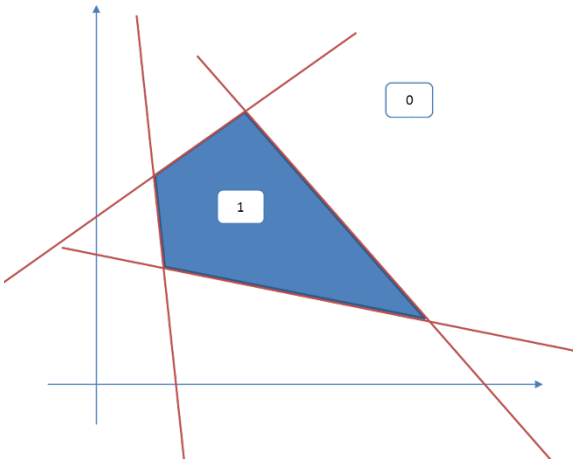


图 6-42　多个线性分类器叠加形成的复杂分类能力

用这 4 条线把四边形围起来，也就是说，必须同时满足 4 个分类器的 1 类标准，才算是我们要约束的 1 类——每条直线的表达式都是形如 $f(x)=wx+b$ 的线性分类器。其实，就是 4 个分类能力非常有限的线性分类器加上几个与、或、非条件而已。

这就是神经网络比其他分类器厉害的地方。神经网络的神经元可以有很多层，每一层中可以有很多个神经元，整个网络中可以有几千甚至几万个神经元。在这种情况下，我们几乎可以描绘出任意的线性不可分模型的情况（如图 6-43 所示）。

图 6-43　丰富的分类

当然，我们在这里只用了一个简单的二维向量来举例。在真正的商用场景中，这些向量通常有几十万个维度或者更多，神经网络的层数也非常多（这就是我们平时所说的深度学习）。随着网络维度和深度的增加，深度神经网络所能描述的分类器的复杂程度也会增加。所以，在传统分类模型中无法通过简单的线性分类器和非线性分类器处理的复杂学习场景（图形、视频、音频等），能够通过海量分类器的叠加来实现——海量的线性分类器和海量的与、或、非条件。

6.8　手写数字识别公开数据集

在了解了深度学习的基本原理后，我们应该开始进行实验了。在开始前，介绍一下手写数字数据集 MNIST，因为我们的实验要拿它“开刀”。

“MNIST”这种辅音连缀一般的前置方式和大写英文的书写风格已经告诉我们，它是一个缩写名词，全称是“Mixed National Institute of Standards and Technology”。MNIST 是一个非常庞大的手写数字数据集，也是网上著名的公开数据集之一。在网上有不少这样的数据集，都是做机器学习和深度学习的研究机构（大学及由政府扶持的公益项目）喜欢拿来进行训练和测试的标准化项目 ①。这些项目对深度学习的初学者来说是非常有帮助的——不管最终是不是要研究强化学习。

这些数据集有两个功能。

一是提供了大量的数据作为训练集和验证集，为学习者提供了丰富的样本信息。这是很宝贵的。要知道，在深度学习领域，要想在一个方面有比较深的研究，除了需要具备一定的网络设计和调优能力，还要有丰富的训练样本。

二是提供了一个在业内相对具有普适性的 Benchmark（参照基准）比对项目。既然大家使用的数据集都是一样的，那么每个人设计出来的网络就都可以在这些数据集上不断进行比较，从而验证网络的识别率。

MNIST 的官方网站为 http://yann.lecun.com/exdb/mnist/。虽说这是一个国际性的大数据集项目的官方网站，但在我看来，它的页面美观程度很一般，如图 6-44 所示。

图 6-44　MNIST 主页

据我观察，很多国外项目的页面都有这种倾向——把核心内容说清楚就可以了，页面好看不

① 关于主流的公开数据集，以及基于这些数据的“刷榜”项目，请参考本书附录 A。

好看是次要的。我猜想，这也许是资深技术人员多年使用 Linux Shell 的“后遗症”吧……开个玩笑。如果读者朋友没有用过 Linux Shell，那就从现在开始尝试吧，因为在我们所有的实验（包括未来的工作）中，有超过 90% 的部分要在 Linux 命令行模式下完成。Linux 命令行模式看上去比这个页面的效果还要粗糙，完全是本着“说明白就行，多一个字浪费”的模式设计的。

在这个页面的下半部分，有一个历年的识别算法对 MNIST 数据集“刷榜”的记录，大家可以参考一下。为了方便读者朋友阅读，我截取了一部分，如图 6-45 所示。

由于 MNIST 只是一个公开的数据集，并非只针对神经网络，任何算法都可以拿它来做测试。在如图 6-45 所示的表格中，我们也能看到如下内容。

- Linear Classifiers：线性分类器。
- k-NearestNeighbors（kNN）：邻近算法。
- Boosted Stumps。
- Non-Linear Classifiers：非线性分类器。
- SVMs：支持向量机。
- Neural Nets：神经网络。
- Convolutional Nets：卷积神经网络。

这几个大的分类器派别，都有一些优秀的代表参与过“刷榜”。在表格中共有四列，含义分别如下。

- CLASSIFIER：分类器名称。
- PREPROCESSING：预处理项。
- TEST ERROR RATE (%)：测试错误率。
- Reference：参考。

分类器名称就不必说了。预处理项的内容类似于注释或者处理技巧，前面几种分类器，例如 Haar Features，就是指加入了 Haar 特征的算法。

我们只看与本书研究内容有关的部分，即神经网络和卷积神经网络。其中提到了两种预处理方式，一种叫作“width normalization”（宽度归一化），另一种叫作“deslanting”。关于归一化的技巧，后面我们会陆续介绍，它对提高识别准确率有很大的帮助。“deslanting”的词根是“slant”，也就是“倾斜”的意思，“deslanting”就是防止数据倾斜的一种手段。

在了解了 MNIST 的背景之后，我们就可以在这个网页上下载如下四个数据文件了。

- train-images-idx3-ubyte.gz: training set images (9912422 bytes)
- train-labels-idx1-ubyte.gz: training set labels (28881 bytes)
- t10k-images-idx3-ubyte.gz: test set images (1648877 bytes)
- t10k-labels-idx1-ubyte.gz: test set labels (4542 bytes)

这四个文件的名字，含义很明确，分别是训练图片、训练标签、测试图片、测试标签。在训练集中有 50000 个数据向量，在测试集中有 10000 个数据向量。

CLASSIFIER	PREPROCESSING	TEST ERROR RATE (%)	Reference
Linear Classifiers			
linear classifier (1-layer NN)	none	12.0	LeCun et al. 1998
linear classifier (1-layer NN)	deskewing	8.4	LeCun et al. 1998
pairwise linear classifier	deskewing	7.6	LeCun et al. 1998
K-Nearest Neighbors			
K-nearest-neighbors, Euclidean (L2)	none	5.0	LeCun et al. 1998
K-nearest-neighbors, Euclidean (L2)	none	3.09	Kenneth Wilder, U. Chicago
K-nearest-neighbors, L3	none	2.83	Kenneth Wilder, U. Chicago
K-nearest-neighbors, Euclidean (L2)	deskewing	2.4	LeCun et al. 1998
K-nearest-neighbors, Euclidean (L2)	deskewing, noise removal, blurring	1.80	Kenneth Wilder, U. Chicago
K-nearest-neighbors, L3	deskewing, noise removal, blurring	1.73	Kenneth Wilder, U. Chicago
K-nearest-neighbors, L3	deskewing, noise removal, blurring, 1 pixel shift	1.33	Kenneth Wilder, U. Chicago
K-nearest-neighbors, L3	deskewing, noise removal, blurring, 2 pixel shift	1.22	Kenneth Wilder, U. Chicago
K-NN with non-linear deformation (IDM)	shiftable edges	0.54	Keysers et al. IEEE PAMI 2007
K-NN with non-linear deformation (P2DHMDM)	shiftable edges	0.52	Keysers et al. IEEE PAMI 2007
K-NN, Tangent Distance	subsampling to 16x16 pixels	1.1	LeCun et al. 1998
K-NN, shape context matching	shape context feature extraction	0.63	Belongie et al. IEEE PAMI 2002
Boosted Stumps			
boosted stumps	none	7.7	Kegl et al., ICML 2009
products of boosted stumps (3 terms)	none	1.26	Kegl et al., ICML 2009
boosted trees (17 leaves)	none	1.53	Kegl et al., ICML 2009
stumps on Haar features	Haar features	1.02	Kegl et al., ICML 2009
product of stumps on Haar f.	Haar features	0.87	Kegl et al., ICML 2009
Non-Linear Classifiers			
40 PCA + quadratic classifier	none	3.3	LeCun et al. 1998
1000 RBF + linear classifier	none	3.6	LeCun et al. 1998
SVMs			
SVM, Gaussian Kernel	none	1.4	
SVM deg 4 polynomial	deskewing	1.1	LeCun et al. 1998
Reduced Set SVM deg 5 polynomial	deskewing	1.0	LeCun et al. 1998
Virtual SVM deg-9 poly [distortions]	none	0.8	LeCun et al. 1998
Virtual SVM, deg-9 poly, 1-pixel jittered	none	0.68	DeCoste and Scholkopf, MLJ 2002
Virtual SVM, deg-9 poly, 1-pixel jittered	deskewing	0.68	DeCoste and Scholkopf, MLJ 2002
Virtual SVM, deg-9 poly, 2-pixel jittered	deskewing	0.56	DeCoste and Scholkopf, MLJ 2002
Neural Nets			
2-layer NN, 300 hidden units, mean square error	none	4.7	LeCun et al. 1998
2-layer NN, 300 HU, MSE, [distortions]	none	3.6	LeCun et al. 1998
2-layer NN, 300 HU	deskewing	1.6	LeCun et al. 1998
2-layer NN, 1000 hidden units	none	4.5	LeCun et al. 1998
2-layer NN, 1000 HU, [distortions]	none	3.8	LeCun et al. 1998
3-layer NN, 300+100 hidden units	none	3.05	LeCun et al. 1998
3-layer NN, 300+100 HU [distortions]	none	2.5	LeCun et al. 1998
3-layer NN, 500+150 hidden units	none	2.95	LeCun et al. 1998
3-layer NN, 500+150 HU [distortions]	none	2.45	LeCun et al. 1998
3-layer NN, 500+300 HU, softmax, cross entropy, weight decay	none	1.53	Hinton, unpublished, 2005
2-layer NN, 800 HU, Cross-Entropy Loss	none	1.6	Simard et al., ICDAR 2003
2-layer NN, 800 HU, cross-entropy [affine distortions]	none	1.1	Simard et al., ICDAR 2003
2-layer NN, 800 HU, MSE [elastic distortions]	none	0.9	Simard et al., ICDAR 2003
2-layer NN, 800 HU, cross-entropy [elastic distortions]	none	0.7	Simard et al., ICDAR 2003
NN, 784-500-500-2000-30 + nearest neighbor, RBM + NCA training [no distortions]	none	1.0	Salakhutdinov and Hinton, AI-Stats 2007
6-layer NN 784-2500-2000-1500-1000-500-10 (on GPU) [elastic distortions]	none	0.35	Ciresan et al. Neural Computation 10, 2010 and arXiv 1003.0358, 2010
committee of 25 NN 784-800-10 [elastic distortions]	width normalization, deslanting	0.39	Meier et al. ICDAR 2011
deep convex net, unsup pre-training [no distortions]	none	0.83	Deng et al. Interspeech 2010
Convolutional nets			
Convolutional net LeNet-1	subsampling to 16x16 pixels	1.7	LeCun et al. 1998
Convolutional net LeNet-4	none	1.1	LeCun et al. 1998
Convolutional net LeNet-4 with K-NN instead of last layer	none	1.1	LeCun et al. 1998
Convolutional net LeNet-4 with local learning instead of last layer	none	1.1	LeCun et al. 1998
Convolutional net LeNet-5, [no distortions]	none	0.95	LeCun et al. 1998
Convolutional net LeNet-5, [huge distortions]	none	0.85	LeCun et al. 1998
Convolutional net LeNet-5, [distortions]	none	0.8	LeCun et al. 1998
Convolutional net Boosted LeNet-4, [distortions]	none	0.7	LeCun et al. 1998
Trainable feature extractor + SVMs [no distortions]	none	0.83	Lauer et al., Pattern Recognition 40-6, 2007
Trainable feature extractor + SVMs [elastic distortions]	none	0.56	Lauer et al., Pattern Recognition 40-6, 2007
Trainable feature extractor + SVMs [affine distortions]	none	0.54	Lauer et al., Pattern Recognition 40-6, 2007
unsupervised sparse features + SVM, [no distortions]	none	0.59	Labusch et al., IEEE TNN 2008
Convolutional net, cross-entropy [affine distortions]	none	0.6	Simard et al., ICDAR 2003
Convolutional net, cross-entropy [elastic distortions]	none	0.4	Simard et al., ICDAR 2003
large conv. net, random features [no distortions]	none	0.89	Ranzato et al., CVPR 2007
large conv. net, unsup features [no distortions]	none	0.62	Ranzato et al., CVPR 2007
large conv. net, unsup pretraining [no distortions]	none	0.60	Ranzato et al., NIPS 2006
large conv. net, unsup pretraining [elastic distortions]	none	0.39	Ranzato et al., NIPS 2006
large conv. net, unsup pretraining [no distortions]	none	0.53	Jarrett et al., ICCV 2009
large/deep conv. net, 1-20-40-60-80-100-120-120-10 [elastic distortions]	none	0.35	Ciresan et al. IJCAI 2011
committee of 7 conv. net, 1-20-P-40-P-150-10 [elastic distortions]	width normalization	0.27 +-0.02	Ciresan et al. ICDAR 2011
committee of 35 conv. net, 1-20-P-40-P-150-10 [elastic distortions]	width normalization	0.23	Ciresan et al. CVPR 2012

图 6-45　MNIST 的“刷榜”记录

MNIST 数据集中的图片，都是手写的数字信息，什么样子的都有。每幅图片中都有一个独立的数字，例如“0”“3”“8”，分辨率为 28 像素 × 28 像素（如图 6-46 所示）。

图 6-46　MNIST 样本示例

6.9　全连接网络

神经网络和深度神经网络的基本构成原理，以及 MNIST 实验数据集的结构，我们已经了解了，下面就开始用 PyTorch 进行基于 MNIST 数据集的相关实验吧。

先来看第一个关于 PyTorch 的全连接网络的例子。请确保在 Ubuntu 环境中下载了本书的代码（如果还没有下载，请参考本书附录 A）。进入目录 feedforward_neural_network/，代码如下。

```
cd ~/DRLPytorch/feedforward_neural_network
```

浏览一下，会看到里面有两个文件，一个是 main-gpu.py，另一个是 main.py。我们打开 main.py 文件。

在阅读代码之前，我们要明确这一次的任务是什么。简单地说，我们希望将一个手写数字图片样本输入网络，在输出端生成一个独热向量，如图 6-47 所示。这个任务可以理解为训练一个能够自动识别手写数字的机器人。

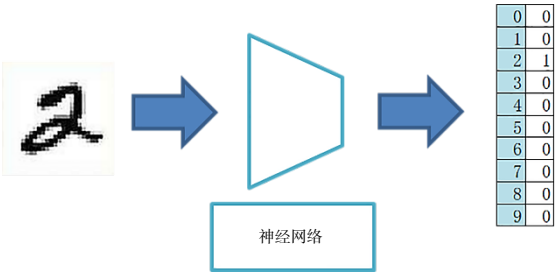


图 6-47　通过全连接网络进行 MNIST 分类

也许有人会说：手写一个程序，用大量的 if…else… 语句也能满足这样的需求。是的，在神经网络技术成熟之前，不得不这么做。不过，不花点时间是没办法写出一个高质量的、可以完成相同任务的代码段的。但是，我们赶上了好时候，只要有样本，机器人就会自己把这个逻辑“学”到。表现出来的形式，也不见得是显式的 if…else… 语句，既可以是一系列函数映射，也可以是一系列矩阵乘法。当然，我们希望它们最终实现的计算逻辑是：通过输入的图片向量的特征，与 $f(x|\theta)$ 中的 θ 进行计算，得到一个我们想要的结果。

6.9.1 输入与输出

对于一个网络来说，最重要的因素莫过于输入、输出、网络结构、损失函数这四个要素。本节我们来了解一下网络的输入与输出。

1. 输入格式

输入是一幅 28 像素 × 28 像素的图片，每个像素都有一个灰度值。这个灰度值是经过归一化处理的，最小值为 0（黑色），最大值为 1（白色），介于其中的小数就是具体的灰度值，如图 6-48 所示。

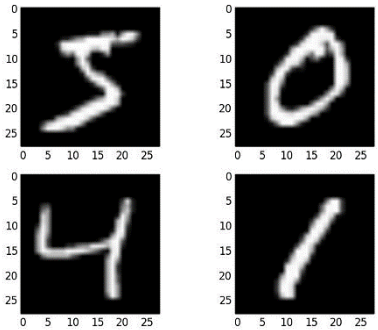


图 6-48 MNIST 样本

2. 输出标签

在训练时，传入网络的是一个独热标签[①]——我们终于讲到它了。在这个例子中，我们希望输出的内容是“2”，但并不是用实数 2 做标签，而是使用表示实数 2 的分类概率的一个 10 维的独热向量

$$[0,0,1,0,0,0,0,0,0,0]$$

做标签。同理，如果是数字 0，就用

① 英文名为“One-Hot Vector”，也可以直接称为“One-Hot”。

$$[1,0,0,0,0,0,0,0,0,0]$$

做标签。

独热向量是一种特性很突出的标签编码方式。在这种编码方式下，你会发现：在这些向量之间，不管发生了什么样的线性变化，它们都不可能彼此变化成另一个有意义的独热向量。例如，[0,0,1,0,0,0,0,0,0,0] 不可能通过任何倍数乘法及加减运算，变成 [1,0,0,0,0,0,0,0,0,0] 这种正交的编码。对于多分类问题（Multi-class Classification Problem）的应用来说，使用这种方式再合适不过了。尤其是对用一个网络来区分猫和狗的图片这样的问题，如果把猫表示成“1”，把狗表示成“2”，那么“言外之意”就是“2 只猫等于 1 条狗”，然而，任何逻辑（包括数理逻辑）都没有办法表达这样的含义，这完全是在误导网络。

6.9.2　代码解读

确定了输入和输出，我们就可以明确网络结构了。看看如下代码。

```
import torch
import torch.nn as nn
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.autograd import Variable


# Hyper Parameters
input_size = 784
hidden_size = 500
num_classes = 10
num_epochs = 5
batch_size = 100
learning_rate = 0.001

# MNIST Dataset
train_dataset = dsets.MNIST(root='./data',
                            train=True,
                            transform=transforms.ToTensor(),
                            download=True)

test_dataset = dsets.MNIST(root='./data',
                           train=False,
                           transform=transforms.ToTensor())

# Data Loader (Input Pipeline)
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

```

```
test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                      batch_size=batch_size,
                                      shuffle=False)

# Neural Network Model (1 hidden layer)
class Net(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out

net = Net(input_size, hidden_size, num_classes)


# Loss and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate)

# Train the Model
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        # Convert torch tensor to Variable
        images = Variable(images.view(-1, 28*28))
        labels = Variable(labels)

        # Forward + Backward + Optimize
        optimizer.zero_grad()  # zero the gradient buffer
        outputs = net(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        if (i+1) % 100 == 0:
            print ('Epoch [%d/%d], Step [%d/%d], Loss: %.4f'
                  %(epoch+1, num_epochs, i+1, len(train_dataset)//batch_size, loss.
data[0]))

# Test the Model
correct = 0
total = 0
for images, labels in test_loader:
    images = Variable(images.view(-1, 28*28))
```

```
79      outputs = net(images)
80      _, predicted = torch.max(outputs.data, 1)
81      total += labels.size(0)
82      correct += (predicted == labels).sum()
83
84 print('Accuracy of the network on the 10000 test images: %d %%' % (100 * correct
/ total))
85
86 # Save the Model
87 torch.save(net.state_dict(), 'model.pkl')
```

1 ~ 33 行可以分成四个部分，具体如下。

- 1 ~ 5 行：导入库。在任何一种编程语言中都有类似的部分。
- 8 ~ 14 行：设置超参数。这些参数是在网络训练前手动定义的，在后面程序会引用它们。
- 16 ~ 24 行：下载训练集和测试集。从这部分代码中可以看出，训练集和测试集是由他人事先写好并封装的，我们只需要调用就可以了。由于这不是本书研究的重点，我们就囫囵吞枣，只关心“黑匣子”外面的内容就可以了。
- 26 ~ 33 行：载入训练集和测试集。拿到训练集和测试集数据的句柄，并把这些下载的文件以程序需要的格式载入。

说到数据格式——是什么格式呢？就是把一幅 28 像素 × 28 像素的图片处理成一个 1 × 784 的向量（即 1 行 784 列的矩阵）。这样处理主要是为了方便地进行全连接网络输入。所谓处理成 1 × 784 的尺寸，就是把每幅图片中的像素“拉直”，使其变成一条横向的“直线”。

1. 网络结构

现在我们看 36 ~ 47 行，就比较容易理解了。通过代换里面的变量关系，可以得到如图 6-49 所示（见彩插）的网络。

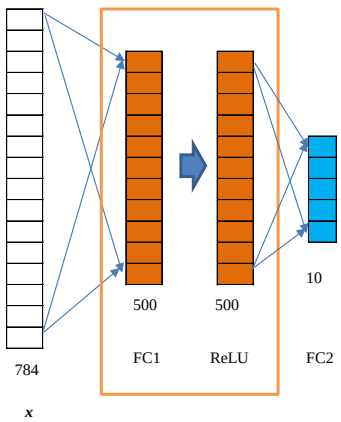


图 6-49　全连接网络示意图

输入为 $\boldsymbol{x}$，一个 784 维（1×784）的矩阵。FC1 是一个全连接层，其中有 500 个神经元，其实是一个 784×500 的矩阵（应该如何计算，会在后面介绍）。ReLU 是激励函数，对于 FC1 输出的 1×500 的矩阵，每一个维度值都被一个非线性激励函数 ReLU 处理过。FC2 也是一个全连接层，相当于一个 500×10 的矩阵。方框中是一个完整的隐藏层，即一个线性函数 $y=\boldsymbol{w}\boldsymbol{x}+b$ 和一个非线性函数 ReLU 叠加的过程。

为了把这个模型的内容讲得更明白，我画了一幅详细的图，如图 6-50 所示（见彩插）。

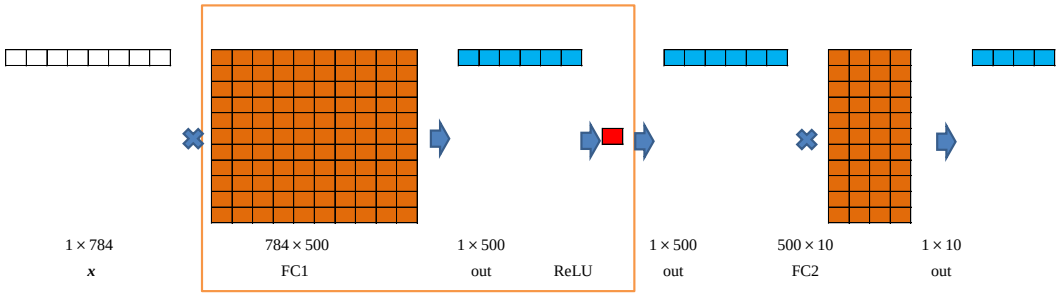


图 6-50　FC 网络结构详解

需要强调一下：图 6-50 和图 6-49 是完全等价的。只不过，图 6-49 是一幅网络连接拓扑图，图 6-50 表示的是相同逻辑下的矩阵乘法和向后传递的过程。

$\boldsymbol{x}$ 是一个 784 维的向量，这是由程序的 26～33 行处理得到的。`nn.Linear` 定义了一个 $y=\boldsymbol{w}\boldsymbol{x}+b$ 函数，只不过这个 $\boldsymbol{x}$ 的大小是 [1,784]。在 PyTorch 中，表示一个矩阵张量（Tensor）的大小时，一般使用“[ ]”的方式。将每个维度的尺寸按顺序填进去，例如 [1,784] 就是指 1×784 的矩阵，依此类推。$\boldsymbol{w}$ 是 [784,500] 的矩阵相乘，加上偏置（Bias）b。

根据矩阵乘法的定义：一个 [1,784] 的矩阵和一个 [784,500] 的矩阵相乘，相当于一个 500 次的循环；每次循环（第 i 次）都是将 [1,784] 的矩阵和 [784,500] 的矩阵的第 i 列做点积，看上去就是两个 784 维的向量，对每两个对应的维度做乘法；最后，把这 784 个值相加，再加上 b，就得到了表达式 $y=\boldsymbol{w}\boldsymbol{x}+b$。

这个计算的概念模型，在 6.2 节中已经讲过了。如果有 500 列，就要循环 500 次，每次都会输出后面这个 [1,500] 矩阵中第 1 行的第 i 个元素，一共输出 500 个，结果为一个 [1,500] 的矩阵。然后，使用 ReLU 函数（就是图 6-50 中红色方块表示的部分）进行处理，表达式为

$$f(\boldsymbol{x})=\max(0,\boldsymbol{x}).$$

在 6.7.1 节中提到过，ReLU 函数像个滤波器。需要注意的是，ReLU 是一个典型的非线性激励函数，一个 $\boldsymbol{x}$ 经过这样的映射后，就会在小于 0 的时候输出 0，在大于 0 的时候输出 $\boldsymbol{x}$ 的值。目前，绝大多数神经网络都会使用 ReLU 激励函数。由于 [1,500] 中的每个元素都通过了 ReLU 函数的处理，它的输出内容仍是一个 [1,500] 的函数，只不过每个元素所对应的都是被 ReLU 函

数处理过的输入值而已。

最后的 500 个元素和 10 个输出节点之间的全连接，在数学上就是一个 [1,500] 的矩阵和一个 [500,10] 的矩阵相乘，输出的结果应该是一个 [1,10] 的矩阵，过程就不再赘述了。

网络结构就是这样，x 通过一系列的计算与映射，会在输出端产生输出结果。当然，在一开始，这个结果是不准确的，因为矩阵的待定系数 w 和 b 都还没有被训练出来。

53 ~ 54 行定义了损失函数。53 行定义了损失函数为交叉熵损失函数 `CrossEntropyLoss()`。54 行声明了用 Adam 优化器（一种工作原理和随机梯度下降很接近的优化器）进行优化。

2. 交叉熵损失函数

什么是交叉熵损失函数呢？在推导损失函数梯度的过程中，我们使用了均方损失函数。均方损失函数通常是用来训练回归问题的，而交叉熵损失函数通常是用来训练分类问题的。

输出的内容是一个 [1,10] 的矩阵，标签的内容也是一个 [1,10] 的矩阵，怎样描述它们之间的误差呢？看这样一个表达式：

$$\text{Loss} = -\frac{1}{n}\sum_{i=1}^{n}[y \ln a + (1-y)\ln(1-a)].$$

我们怎么理解交叉熵的含义呢？可以这么想：当一个样本通过网络产生一个 [1,10] 的标签拟合值时，必然和它期望的标签值（那个独热编码）有差距。

我们一起来看一个示例。输入一个“2”的图片样本，这个样本 x 通过网络输出了一个矩阵

$$[0.5,0.5,0,0,0,0,0,0,0,0],$$

期望的标签为

$$[0,0,1,0,0,0,0,0,0,0].$$

不要以为这么离谱的标签网络输出不了。网络的参数 θ 是“随意”初始化的，因此不可能一上来就满足如此精确的计算需求。对每一位的分类误差进行求和运算，就得到了当前网络分类器的分类能力和期望的分类能力之间的差距。

在上面的公式中，y 就是期望的标签值。当 $y=1$ 时，$1-y=0$，于是后面那一项就不存在了，这时期望 $\ln a = 0$。这里的 a 是指对应位置的输出值，也就是拟合出来的标签值（因为比 0 大的都是误差值）。如果 $\ln a$ 等于 0，那么 a 必须等于 1，也就是所谓的标签 y。拟合出来的对应位置的输出值最好为 1，因为此时的误差值为 0。后面那一项则是反过来的情况，就是当 $y=0$ 时生效。大家可以自己理解一下，看看是不是同一个道理。

那么，刚刚这个样本产生的损失值是多大呢？大概是

$$\text{Loss} = -[\ln(1-0.5) + \ln(1-0.5) + \ln 0 + \ldots].$$

其中，省略号代表的 7 项都是 0。

损失函数的值就是所有的训练数据（在一个 Epoch 中）在每个样本上的误差平均值，那么优化的方向就是：使得整个网络中的待定系数朝着减小这个误差值的方向调整。例如，对那些应该拟合出“3”的，使对应的“3”的向量位置为 1，其他位置为 0。这就是训练中参数优化的方向，即找“碗底”的过程。

57 ~ 72 行的部分似曾相识。65 行正向传播一幅图片，这时会得到拟合值。66 行计算损失函数的值。67 行进行一次反向传播。68 行进行一次优化，使整个网络的 w 和 b 都向着减小误差的方向挪动一轮。外面是两个循环，一个是 Epoch 的数量，另一个是对 `train_loader` 中的每个元素（图片、标签）进行循环。

“Epoch”是机器学习中一个用来称呼样本量的名词。在机器学习中，如果准备了 50000 个样本，当这 50000 个样本都通过网络产生了一次正向传播的时候，就叫作一个 Epoch。在本例中，`batch_size=100`，也就是一次随机挑选 100 个样本进行训练，再进行一次梯度下降，那么，每处理一批（Batch）数据，就叫作一个 Step。如果这 50000 幅图片要完整地通过一个网络完成一个 Epoch，就需要 500 个 Step，而且是那种 Batch 从全部样本中抽取且不再放回的。具体的训练细节当然可以由我们自己决定，原则就是希望这些样本都能等权重地在训练中发挥作用，也就是让每个样本都能有同等的机会进行正向传播并产生相应的损失值。

74 ~ 82 行是测试部分，其中靠后的部分是在测试集上进行验证，以防出现过拟合。77 行的循环就是遍历 `test_loader` 这个测试集。80 行中有一个 `torch.max()`函数，用于返回指定列中值最大的那个元素并返回索引值。如果打印出 `labels` 和 `predicted`，你就会发现，它们都是用索引值（1、2、3……）表示的，而不是独热向量。后面几行用于进行统计，计算预测正确的样本数量。

一旦明确了交叉熵损失函数的意义，我们就会发现，图片分类将变成一项技术含量很低的工作。只要能区分“1”“2”“3”“4”，就能区分猫、狗、大象，也就能区分其他任何可以用类似理论解释的模型和场景（如图 6-51 所示）。

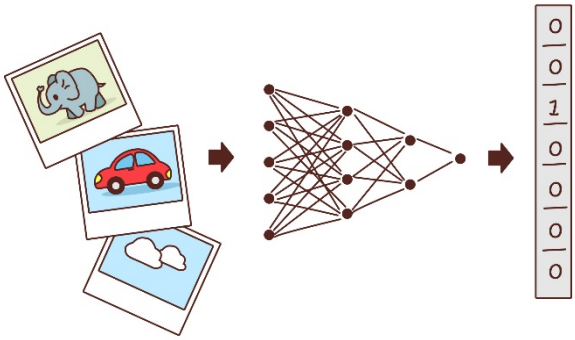


图 6-51　多分类问题都可以使用独热向量做标签

6.9.3　运行结果

运行如下命令。

```
python main.py
```

如果环境配置没有问题，就可以正确运行这个命令。如图 6-52 所示，在这个数据集中的训练效果还不错，在测试集中正确率为 97%。

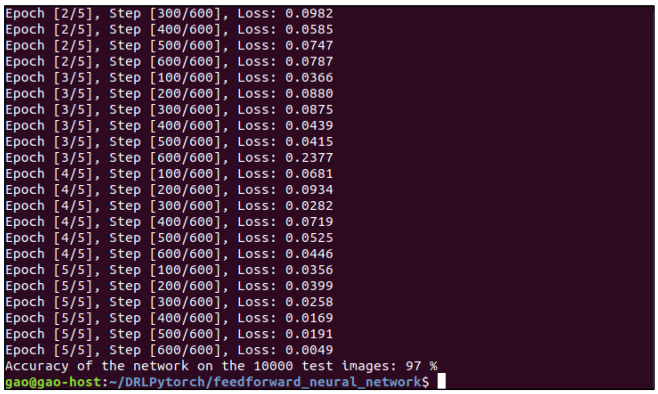


图 6-52　用 FC 训练 MNIST 的正确率

这样，一个用 2 层全连接网络实现 MNIST 分类的功能就完成了——程序只有不到 100 行，却能实现这么复杂的功能。如果使用非机器学习的方法，手动编写代码来实现手写图片自动分类功能，过程可能就比较痛苦了。想想本节开头提到的各种 if…else… 语句吧……还是挺闹心的。而现在，这些都由神经网络实现了，我们需要做的只是把数据找到、把网络搭好、把损失函数定义好，其他的交给计算机就可以了——科技改变人生。

6.10　卷积神经网络

在 6.9 节中，我们用全连接网络实现了一个简单的手写数字识别功能。在这一节中，我们来看另一种方式——PyTorch 的卷积神经网络的分类模型怎么定义。

6.10.1　代码解读

进入目录 ~/DRLPytorch/convolutional_neural_network/，打开 main.py 文件，其内容如下。

```
1 import torch
2 import torch.nn as nn
3 import torchvision.datasets as dsets
4 import torchvision.transforms as transforms
5 from torch.autograd import Variable
```

```
 6
 7
 8 # Hyper Parameters
 9 num_epochs = 5
10 batch_size = 100
11 learning_rate = 0.001
12
13 # MNIST Dataset
14 train_dataset = dsets.MNIST(root='./data/',
15                          train=True,
16                          transform=transforms.ToTensor(),
17                          download=True)
18
19 test_dataset = dsets.MNIST(root='./data/',
20                         train=False,
21                         transform=transforms.ToTensor())
22
23 # Data Loader (Input Pipeline)
24 train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
25                                      batch_size=batch_size,
26                                      shuffle=True)
27
28 test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
29                                     batch_size=batch_size,
30                                     shuffle=False)
31
32 # CNN Model (2 conv layer)
33 class CNN(nn.Module):
34    def __init__(self):
35        super(CNN, self).__init__()
36        self.layer1 = nn.Sequential(
37            nn.Conv2d(1, 16, kernel_size=5, padding=2),
38            nn.BatchNorm2d(16),
39            nn.ReLU(),
40            nn.MaxPool2d(2))
41        self.layer2 = nn.Sequential(
42            nn.Conv2d(16, 32, kernel_size=5, padding=2),
43            nn.BatchNorm2d(32),
44            nn.ReLU(),
45            nn.MaxPool2d(2))
46        self.fc = nn.Linear(7*7*32, 10)
47
48    def forward(self, x):
49        out = self.layer1(x)
50        out = self.layer2(out)
51        out = out.view(out.size(0), -1)
52        out = self.fc(out)
53        return out
54
```

```
55 cnn = CNN()
56
57
58 # Loss and Optimizer
59 criterion = nn.CrossEntropyLoss()
60 optimizer = torch.optim.Adam(cnn.parameters(), lr=learning_rate)
61
62 # Train the Model
63 for epoch in range(num_epochs):
64     for i, (images, labels) in enumerate(train_loader):
65         images = Variable(images)
66         labels = Variable(labels)
67
68         # Forward + Backward + Optimize
69         optimizer.zero_grad()
70         outputs = cnn(images)
71         loss = criterion(outputs, labels)
72         loss.backward()
73         optimizer.step()
74
75         if (i+1) % 100 == 0:
76             print ('Epoch [%d/%d], Iter [%d/%d] Loss: %.4f'
77                    %(epoch+1, num_epochs, i+1, len(train_dataset)//batch_size,
loss.data[0]))
78
79 # Test the Model
80 cnn.eval()  # Change model to 'eval' mode (BN uses moving mean/var).
81 correct = 0
82 total = 0
83 for images, labels in test_loader:
84     images = Variable(images)
85     outputs = cnn(images)
86     _, predicted = torch.max(outputs.data, 1)
87     total += labels.size(0)
88     correct += (predicted == labels).sum()
89
90 print('Test Accuracy of the model on the 10000 test images: %d %%' % (100 *
correct / total))
91
92 # Save the Trained Model
93 torch.save(cnn.state_dict(), 'cnn.pkl')
```

我们来解析一下这个文件的内容。其实还是很简单的，仍然是一个不到 100 行的小文件。30 行之前的部分，在 6.9 节中已经进行了解释：导入库；设置超参数；使用官方提供的 API 获取 MNIST 数据的训练集和测试集。这个文件的核心是 32 ~ 55 行，这部分定义了一个 CNN 类。

卷积神经网络（Convolutional Neural Network，CNN）因为在网络中有卷积核（Convolution Kernel）这样一个功能组件而得名。

由于卷积神经网络在工程中有比较好的表现效果，近年来的网络模型几乎离不开卷积核。卷积核究竟是什么，又有什么作用呢？下面进行简单的解释。

卷积核通常是一个正方形的框，尺寸可以是 1×1、2×2、3×3 等，如图 6-53 所示（见彩插）。当然，在一些特殊的场景中，卷积核也可以是非正方形的。

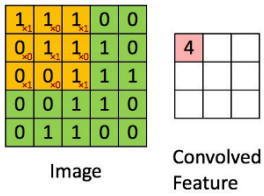


图 6-53　图片的输入和卷积特征（1）

卷积核所做的事情就是把前一层输入的数据特征提取出来。所谓特征提取，就是通过一个线性计算映射和一个激励函数（可选），让它在后一层成为一个数值形式的激励值。在如图 6-53 所示的卷积核中，如果输入一个 5×5 的图形，图形中的每个像素都是一个具体的数字，一般我们还是喜欢用 R、G、B（红、绿、蓝）三通道的数值来表示。具体表现出来，就是一幅图片会分成以红色、绿色、蓝色通道输入的三幅不同的图片，每个像素的取值为 0～255。在这里，为了表述方便，只用 0 和 1 来表示，也可以直接认为 0 代表黑色、1 代表白色——是的，一幅黑白图片。另外，用一个 3×3 的卷积核对它进行 $y = wx + b$ 的卷积（线性计算），其中

$$w = [1,0,1,\\ 0,1,0,\\ 1,0,1],$$

那么，卷积核在进行特征提取的时候，就会如图 6-53 所示，先按位进行乘法运算，再进行加法运算，即

$$y = 1\times 1 + 0\times 1 + 1\times 1 + 0\times 1 + 1\times 1 + 0\times 1 + 1\times 0 + 0\times 0 + 1\times 1 + 0 = 4.$$

这个“4”会被“投射”到其后方的 Feature Map 中去，也就是图 6-53 中 Convolved Feature 部分左上角的那个点。

如下问题需要我们注意。

- 这不是矩阵乘法，而是两个向量做点积，即向量对应位置的值相乘再相加。
- 在这个例子中，$w = [1,0,1,0,1,0,1,0,1]$，$b = 0$，用这样的卷积核进行卷积操作。
- 在一次扫描后，卷积核会从左到右、从上到下扫描整幅图片，并将每次计算中产生的结果放到后面对应的 Feature Map 中去。在这个例子中，会产生一个 3×3 的 Feature Map。
- 如果一个 Feature Map 要与前面输入的图片的尺寸保持一致，通常需要在边缘补 0，这种操作叫作 Padding。如图 6-54 所示，一个 32×32 的输入，被 Padding 操作处理成了 36×36 的

大小（在四个边上分别补了 2 个 0）。补 0 的目的，一是保证通过卷积操作得到的 Feature Map 和输入的图片尺寸相同，二是使边缘的数据被扫描的次数有所增加。否则，可以想象，图片边缘的像素点比起中间的像素点，被扫描的次数会有所减少，而这其实是一种信息的丢失。

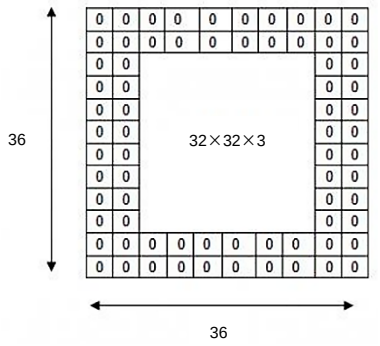


图 6-54　Padding 示意图

在以上第三个注意事项中，扫描是可以“跳跃”进行的，也就是说，可以一次移动 1 个像素后进行扫描，也可以一次移动 2 个或更多个像素。移动 1 个像素，就是 Striding=1，这是扫描最为密集的方式。Striding 的值可以为 2 或者更大，值越大，则跳过的信息越多，扫描的速度也就越快，但会牺牲扫描提取的信息量。另外，Feature Map 的尺寸会受 Striding 的影响，Striding 的值越大，Feature Map 的尺寸就越小。

以如图 6-54 所示的 Padding 为例，在 Striding=1 的情况下，一次向右移动 1 个像素再扫描，将产生一个新的“投射”值，那么，在这一行就会产生 3 个值的“投射”值。“走”到头，然后转行，纵向移动 1 个像素，继续扫描。依然进行和第一行一样的扫描操作，又会产生 3 个值。如此这般，卷积核会产生一个 3×3 的 Feature Map。每个值应该为多少，需要根据 $y = wx + b$ 的逻辑来计算。

这就是一个卷积核对一个通道进行扫描的完整过程。卷积完成，如图 6-55 所示（见彩插）。

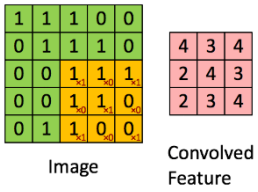


图 6-55　图片的输入和卷积特征（2）

在 PyTorch 中，如果要使用卷积核对一幅图片进行扫描，通常需要定义卷积核的大小和数量。大小是指卷积核的尺寸，也叫作“Kernel Size”。数量是指用多少个这样的卷积核对图片进行

扫描。卷积核的数量也叫作通道数（Channel）。在训练的时候，就是在让 PyTorch 框架帮我们用梯度下降的方式找出各个卷积核的 w 各自取什么值才能让模型有足够好的表现。一般来说，卷积神经网络需要的参数的数量比全连接网络更少，同时训练速度更快、泛化能力更强。

我们再来看一下模型的定义。

- 36 ~ 40 行：定义第一个卷积层。
- 37 行：输入为 1 通道，输出为 16 通道（即输入一幅 1 通道的图片，有 16 个卷积核来扫描它）。卷积核为 5 × 5。Padding=2 就是在四边各补 2 个 0。
- 38 行：对 37 行的结果进行批归一化（Batch Normalization），目的是防止训练过程中各网络层的输入落入饱和区域而导致梯度消失。在深度学习中使用的批归一化，与在传统机器学习中用的归一化（Normalization）的目的不太一样。使用批归一化的目的是防止梯度消失，归一化的作用会在 6.12.3 节单独说明。
- 39 行：使用 ReLU 激励函数处理 38 行输出的结果。
- 40 行：对 39 行输出的结果进行降采样（Down Sampling），这里使用的是 Max Pooling。

所谓 Max Pooling，就是如图 6-56 所示（见彩插）的操作。

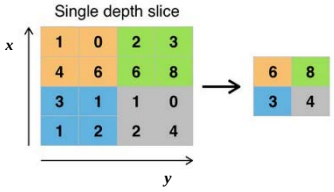


图 6-56　Max Pooling 示意图

输入一幅图片或者一个 Feature Map，在相邻的 $n \times n$ 的方框里，挑选最大的一个来输出结果。如图 6-56 所示，一幅 4 × 4 的图片，在通过 2 × 2 的 Max Pooling 层之后，就变成了一幅 2 × 2 的图片。

这种降采样不仅可以取最大值（用 Max Pooling），也可以取平均值（用 Average Pooling）。其目的都是通过用某一个值来代表前一层的若干邻近值，相当于进一步进行特征提取和降维操作。Max Pooling（或者 Average Pooling）的尺寸可以也可以是 2 × 2、3 × 3、4 × 4 等，这要根据实际需要来确定。在这个例子中，使用的是 2 × 2 的 Map Pooling。

- 41 ~ 45 行：输入为 16 通道，输出为 32 通道，卷积核为 5 × 5，Padding=2。在这里仍然进行了批归一化（通过 ReLU 函数和 2 × 2 的 Max Pooling）。

16 通道的输入和 32 通道的输出，可以理解为，对前面通过卷积得到的 16 个 Feature Map，用 32 个卷积核进行卷积，重新得到了 32 个 Feature Map。

- 46 行：最后一层。这是一个线性全连接层，输入为 7 × 7 × 32，输出为 10 个节点，其实就是将输出 $[1,7 \times 7 \times 32]$ 尺寸的向量作为 $\boldsymbol{x}$（一个 $[1,1568]$ 的张量）和一个 $[7 \times 7 \times 32,10]$

的 $\boldsymbol{w}$ 相乘，再加 b（b 是一个 [1,10] 的张量）。

这个 $7\times7\times32$ 是怎么得到的呢？我们想想看，MNIST 图片的大小是 28 像素 × 28 像素，如果直接用 5×5 的卷积核来扫描，仍然会得到 28×28 的 Feature Map，通过 2×2 的 Max Pooling，会得到 16 个 14×14 的 Feature Map。用 5×5 的卷积核，取 Padding=2，处理 14×14 的 Feature Map，仍然会得到 14×14 的 Feature Map，通过 2×2 的 Max Pooling，输出的尺寸为 7×7。因为卷积核有 32 个，所以尺寸就是 $7\times7\times32$（即 1568 个特征值）。

整个卷积神经网络，如图 6-57 所示（见彩插）。

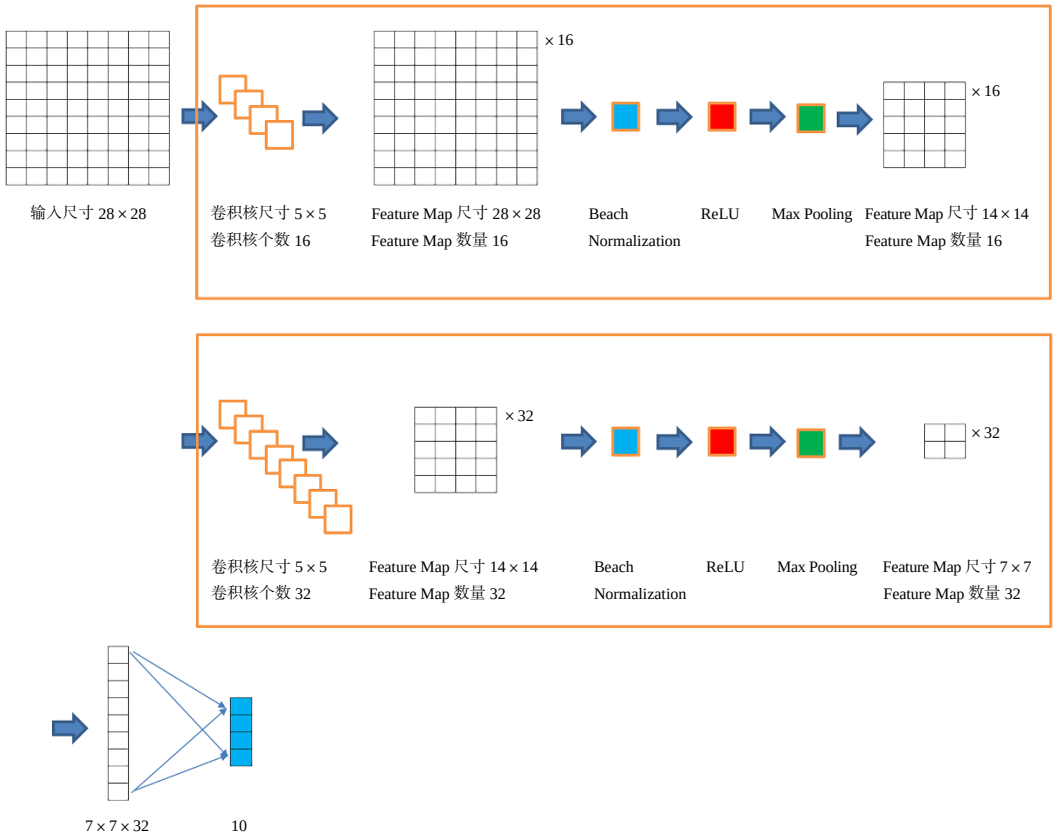


图 6-57　卷积神经网络示意图

- 58 ~ 60 行：损失函数就是交叉熵损失函数（和在全连接网络中使用的损失函数一样）。
- 61 ~ 93 行：训练、测试、验证及模型保存部分。这些步骤在前面的实验中都涉及了，在这里不再赘述。

执行如下命令。

```
python main.py
```

大约有 98% 的正确率。尽管精度提高了，但训练速度比在全连接网络中慢了得多。

6.10.2 理解卷积神经网络的结构

我们能够在网上找到很多讲解卷积神经网络的博客或者免费资料。虽说每个人对卷积神经网络都有自己的理解，但这些内容基本都来自国外的高校教科书或者公开课，不少说法都显得“文绉绉”的。鉴于卷积神经网络是深度学习、深度强化学习中不可或缺的一个环节，我想，还是对它进行简单的补充说明吧。

图 6-58（见彩插）“可视化”地描述了卷积神经网络的工作过程。我觉得，只要把这幅图看懂了，其他任何卷积神经网络及其相关应用，我们都能理解个八九不离十。

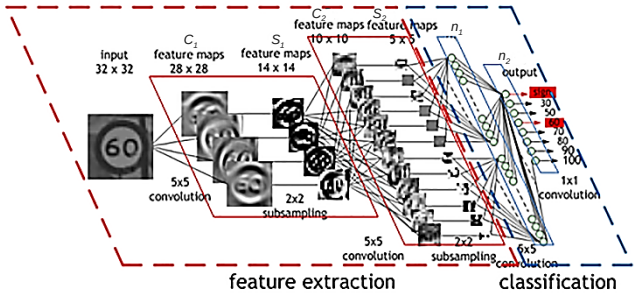


图 6-58 卷积神经网络结构

我们希望将一幅 32 像素 × 32 像素的灰度图片输入卷积神经网络，让网络自动输出这幅图片的分类。从输入来看，图片中有一个圆圈，里面有一个阿拉伯数字 60。在右侧的蓝色虚线框中，输出的是 30、50、60、70、80、90、100 这样的向量。这就像自动驾驶机器人中的一个小模块，模块的功能就是希望在输入一个限速图标时识别这是哪种限速图标、限速多少。

输入的 32×32 的图标，经过 4 个 5×5 的卷积核的计算，会在 C_1 层产生 4 个 28×28 的 Feature Map。从 Feature Map 的尺寸来看，显然没有进行 Padding 操作。关于在卷积层中要不要进行 Padding 操作，需要根据输入图像的边缘携带的信息的特征及其是否足以影响分类的准确率来决定。在这个例子中，表示限速的“60”在图像的中间，周边是圆圈及外侧限速牌等背景信息，对分类没有什么帮助。

C_1 层的 4 个 28×28 的 Feature Map，通过一个 2×2 的降采样处理后，会变成一个新的 14×14 的 Feature Map——无疑，通过了 Pooling 层。这个地方可以是 Max Pooling，也可以是 Average Pooling，在不同的业务场景中，两种 Pooling 各有特点。这种降采样的作用，就是用 1 个点的信息代替 4 个点的信息，是一种典型的降维操作，表现了一种典型的以一定的信息损失为代价换取空间和运算时间的取舍态度。

使用 Max Pooling 的理由应该说是充分的：如果用 4 个点（2×2）、9 个点（3×3）或更多的点为代表来表示原先的若干个点，那就取一个最大的，这个值应该是对网络后层刺激最大的值。这有点像各地选拔运动员参加全运会，把每个地区代表队中最厉害的运动员选出来，然后在赛场上“PK”。

另一种叫作 Average Pooling 的方法也是有理有据的。它选用 4 个点或者几个临近点中的平均值作为代表来实现降维。根据统计学中的相关理论，要想表征一个群体的特性，最大值也好，最小值也罢，都不能作为代表，而有一定代表作用的用中位数或者平均值都有合理性。要知道，在我们上学的时候，老师的教学水平在某种程度上是可以用学生的平均分来评价的，就算是去掉最高分和最低分之后再做统计，也是取平均分的。因此，将平均值作为代表的理由也是非常充分且有依据的。

在深度学习框架中，通常会对这两种方式都进行保留，这就是本例中 S_1 层采取的思路了。Max Pooling 和 Average Pooling 都有自己的道理，而我们在做具体项目的时候，还是要仔细研究一下——要么通过自己做实验来证明某种假设更为正确，要么根据以前论证过这些观点的论文来进行取舍。

C_2 层和 S_2 层，分别相当于又一次进行了多通道卷积核的特征提取和进一步的降维操作。也就是说，通过 12 个卷积核对前面的 Feature Map 进行处理，输出 12 个新的 Feature Map，再分别进行降采样，最后输出 12 个 5×5 的 Feature Map。

到这一步，已经完成两次特征提取了。对于比较复杂的模型需求来说，有时可能会设计数十个卷积层来进行比较丰富的特征提取，如图 6-59 所示。

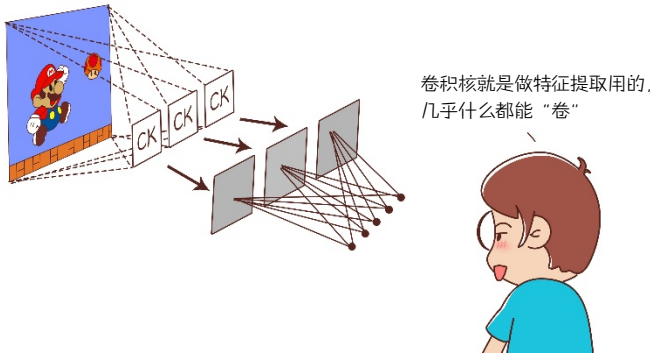


图 6-59　卷积核数量多则特征提取能力强

在如图 6-58 所示的蓝色虚线框里面有两层。前面一层叫作 n_1，用 5×5 的卷积核进行卷积，得到了若干个 1×1 的 Feature Map——数量蛮多的。后面一层采用一个 7 通道 1×1 的卷积层，提取了预测的独热向量的概率表示，这个操作从图上是看不出来的（貌似是没用的）。这一层在一些经典的资料中可能会用全连接网络实现。不过，用什么方法实现并不重要，虽说深度学习建模有

一定的套路和规矩，但只要最终的实验效果是好的，即便与经典理论有悖，也无须挂怀，因为这很可能是一项创新理论。

卷积神经网络是因为卷积层的存在而得名的。对于现在的深度神经网络模型来说，几乎每个模型中都有卷积层的存在，所以，通常不会有人强调这个网络是否是卷积神经网络，而是具体描述这个网络有多少层、每层是由什么功能组件构成的等。简言之，卷积层的作用是特征提取，池化层的作用是降维，从而降低模型的训练难度。

6.10.3 卷积核的结构

在 6.10.2 节的例子中有一个细节，不知道大家有没有注意到：C_2 层相当于用 12 个卷积核对前面的 Feature Map 进行了卷积，输出了 12 个新的 Feature Map；输入不是 1 通道，而是 4 通道。这个卷积层是怎么实现输入为 4 通道、输出为 12 通道的呢？在这里到底定义了几个卷积核？

关于卷积核的数量，不必纠结。有几个通道的输出，就定义几个卷积核——没错，12 个。但是，12 个卷积核如何处理 4 通道的输入呢？平均一个输入通道处理 3 个卷积核吗？当然不是，因为如果定义 11 个或者 13 个卷积核，这个网络仍然能够正常工作（不信的话可以自己修改代码试试看），这压根儿不是一件与整除和平均有关的事情。那么，是对前面一层各通道的 Feature Map 进行了平铺处理吗？也不是，因为如果分别打印这个卷积层前面和这个卷积层后面的 Feature Map 的尺寸，你就会发现，这两个 Feature Map 除了通道数量不一样外，每个通道的尺寸和大小都是一样的（这也说明不是平铺）。

对一幅 5×5×3 的图片（5×5 大小的 3 通道图片，也可以说，3 个 Feature Map，每一个的尺寸是 5×5），用 2 个卷积核（2 通道的卷积层）进行卷积运算的过程，如图 6-60 所示（见彩插）。可以看出，增加了 Padding。这 2 个卷积核的参数位置，和 6.10.1 节卷积神经网络文件中的 37 行

```
nn.Conv2d(1, 16, kernel_size=5, padding=2)
```

中的“16”含义相同。需要注意的是，在这里定义了卷积核的数量为 2，但这个“2”包含的卷积核是 3×3×3 的，也就是说，3×3 是卷积核的尺寸（`kernel_size`），另一个 3 是由前面输入的通道数自动决定的。

W0 有 3 个 3×3 的卷积核矩阵，这是 3 套不同的 $y=\boldsymbol{w}\boldsymbol{x}+b$，$b$ 是共享的，但每个卷积核的 $\boldsymbol{w}$ 不一样。W1 的尺寸和 W0 一模一样，也是一个 3×3×3 的卷积核，只是具体的参数值不同。这两套卷积核，一共有 6 个矩阵，各“卷”各的，只不过 W0 内部的 3 个矩阵在各自“卷”完后还要加在一起。在这个计算瞬间，W0 计算的内容是

$$\begin{aligned}&2\times 1+1\times 1+1\times 1=4\\&1\times(-1)=-1\\&1\times(-1)=-1.\end{aligned}$$

这是 3 套不同的 wx。把它们加起来，再加上共享的偏置 $b=1$，得到

$$4-1-1+1=3.$$

这就是输出的 2 个通道中第一个通道的第 1 列第 2 个值的计算方法了。

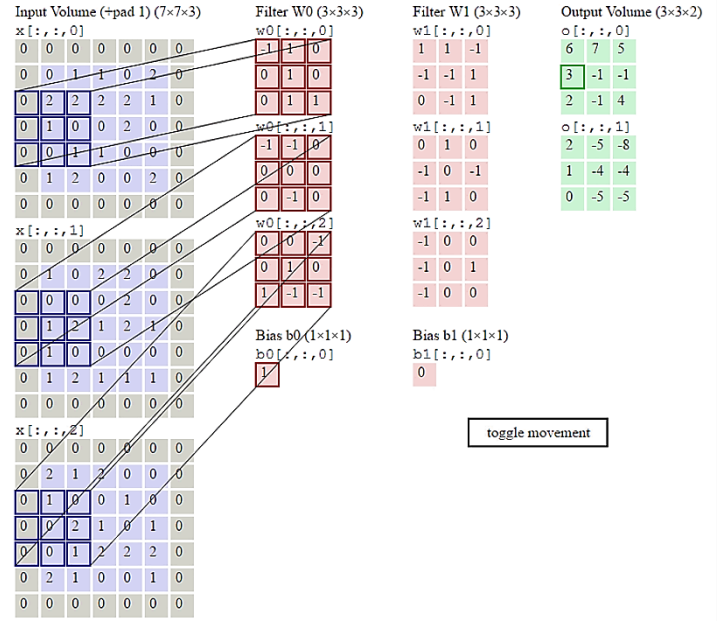


图 6-60　多通道卷积核的实现

其他位置依此类推，包括第二个通道 W1 所得到的数据，也是同理可得的。为了节省篇幅，我们就不再复现了。

6.11　循环神经网络

在 6.10 节中，我们用一个不到 100 行的 PyTorch 程序实现了一个卷积神经网络。其实，仔细算起来，这个网络有 4 层，只不过在网络定义中，像 Pooling 层和激励函数这样的层，因为没有待定系数，所以不单独作为一层也是可以的。而且，PyTorch 支持把这些“组件”声明在同一个层（Layer）中。在这一节中，我们来学习深度学习中的另一个重要概念——长短期记忆（Long Short-Term Memory，LSTM）网络。

LSTM 网络是循环神经网络（Recurrent Neural Networks，RNN）的一种形式，也是当前较为流行的循环神经网络的实现方式。在这里有必要简单讲一下 RNN 和 LSTM 网络的优势。

RNN 与 FF 网络及 CNN 网络在结构上有明显的不同。FF 网络通常用全连接层来实现，CNN 则以带有卷积核的网络层来组合实现一个网络。

LSTM 网络是用循环层实现的。循环层有一个特点，就是把前一次（时序上）输入的内容或

者其中间的激励值，以及当前这一次的输入值，一起作为网络的输入，如图 6-61 所示。

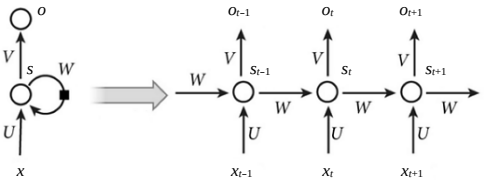


图 6-61　RNN 时序展开后的结构

采用这种结构的好处也非常明显：如果在时序上，前一次的内容对这一次的输入有影响，那么这种影响的关系是可以被学习的。而对这种时序上的关系，FF 网络和 CNN 天生就不具备学习能力。

有这样一个例子：

今天的天气非常（ ）

如果此时输入拼音“zao”，应该预测出“早”“糟”还是其他字呢？根据上下文，“糟”更为通顺（这种内容就是通过大量的数据样本训练得到的）。人能够进行推理，也是因为见过很多样本，才能推断出那些更为接近经验的正确结论。现在，我们想让机器人具备这样的学习能力。在时序上根据前面的内容推断现在应该产生的输出值，是 RNN 的主要功能。因此，根据 RNN 本身的特性，使用它来训练一个对话机器人（输入一个句子，期望机器人输出一个句子；输入一个英文句子，期望机器人输出一个中文句子）是很适合的。

不过，最早一代 RNN 使用的 BPTT 算法[①]已经过时了，现在主流的算法倾向于使用 LSTM 网络或者 GRU[②]。

6.11.1　网络结构

LSTM 网络的结构比较复杂，如图 6-62 所示。一个 LSTM 网络通常由一层或者两层 LSTM 单元组成，每一层中的单元可以首尾相接，在时序列上，输入可以从前向后传递。

对于一个输入的序列 X_i 来说，X 的值可能会影响一个在时间上或者空间上距离比较远的 h_j 的输出，训练的过程就是将量化关系“学”到。

在训练的时候，LSTM 可以如图 6-63 所示，组成一个很大的方阵。其中，除了输入层和输出层分别对应于 X_t 和 h_t，中间部分是一个个 LSTM 单元。需要注意的是，LSTM 单元中的参数非常多，这除了会占用较多的内存，还容易产生过拟合的问题。所以，在实际应用中，极少看到多

① BPTT（Back Propagation Through Time），一种用于在 RNN 结构中进行误差传递的古老算法。
② GRU（Gated Recurrent Unit），RNN 的一种结构单元，有比较广泛的应用。

层的 LSTM。在我做过的项目中，大多只使用一层 LSTM，两层 LSTM 就已经非常多了，再多的我还没见过。

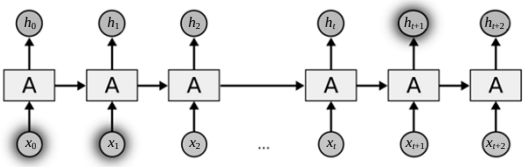


图 6-62　LSTM 网络结构

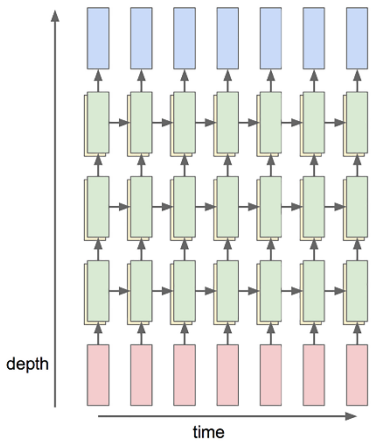


图 6-63　深度 LSTM 网络

我们来解剖一下这个单元，看看它有什么特别的地方（如图 6-64 所示）。

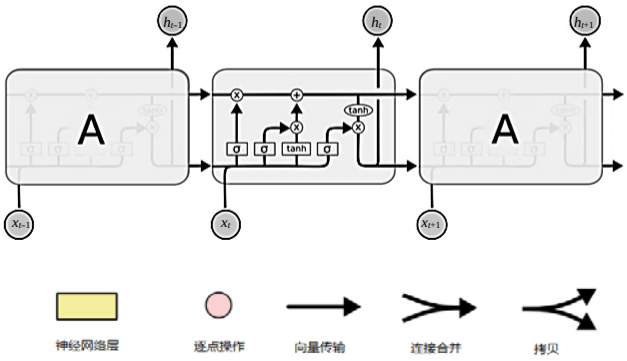


图 6-64　LSTM 说明图例

一个个 LSTM 单元看上去是首尾相接的。在同一层中，前面单元的输出会作为后面单元的输

入，前一层的输出会作为后一层的输入——就像一个个“黑匣子”。这样做的好处是，我们只关心接口部分就可以了。下面，我们具体看看一个单元中都有什么。

如图 6-65 所示，从左到右，会有一个向量进行传输。它从单元的左侧输入，称作 $\boldsymbol{C}_{t-1}$；从单元的右侧输出，称作 $\boldsymbol{C}_t$。在这里，只有两个交互的部分，一个是左边的“乘号”，另一个是右边的“加号”。先说“加号”部分，就是普通的向量线性叠加。左边的“乘号”是一个乘法器，这个操作相当于左侧的 $\boldsymbol{C}_{t-1}$ 进入单元后，先被一个乘法器乘以一个系数，再线性叠加一个值，然后从右侧输出。

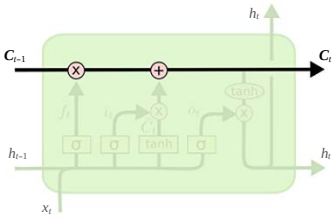


图 6-65　LSTM 单元（1）

如图 6-66 所示，通过乘法器乘以的这个系数的来源是：左侧的 h_{t-1} 和下面输入的 x_t，经过连接操作，再通过一个线性单元和一个 σ（也就是 Sigmoid 函数），生成一个 0 到 1 之间的数字。它的表达式是这样的：

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f).$$

这个部分就是一个“忘记门”（Forget Gate）。

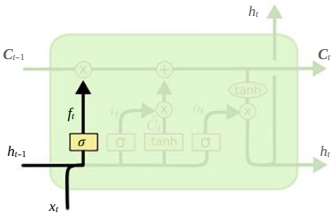


图 6-66　LSTM 单元（2）

所谓“忘记”，就是指这个相乘的过程。如果 Sigmoid 函数的输出为 1，表示完全“记住”；如果 Sigmoid 函数的输出为 0，表示完全“忘记”；中间的值，就是“记忆”的比例（或者说“忘记”的比例）。W_f 和 b_f 要作为待定系数进行训练和学习。

如图 6-67 所示，这里有两个小的神经网络层。一个是我们熟悉的 σ 部分，表达式为

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i).$$

旁边的 tanh 也代表一个神经网络层，表达式为

$$\tilde{\boldsymbol{C}}_t = \tanh(W_C[h_{t-1}, x_t] + b_C).$$

这个 tanh 函数在介绍激励函数的时候提到过一次，它可以把一个值映射到 –1 和 1 之间。这里的 W_C 和 b_C 也要通过训练得到。

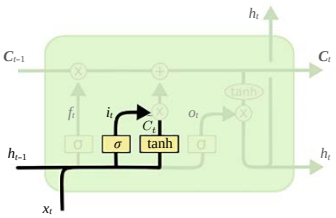


图 6-67　LSTM 单元（3）

在这之后，如图 6-68 所示，前一次传递过来的向量 $\boldsymbol{C}_{t-1}$ 会和 $\tilde{\boldsymbol{C}}_t$ 进行线性叠加。

$$\boldsymbol{C}_t = f_t \cdot \boldsymbol{C}_{t-1} + i_t \cdot \tilde{\boldsymbol{C}}_t.$$

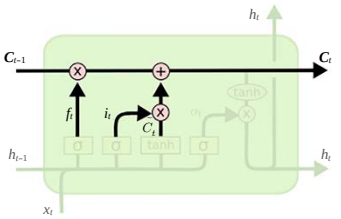


图 6-68　LSTM 单元（4）

到这里，其实决定了本次输出的 $\boldsymbol{C}_t$ 究竟有多少采纳了本次输入的信息、有多少采纳了上一次遗留下来的信息。在 NLP 应用的语言模型中，就是前一个主语在当前是否应该被遗忘，新的主语是否应该代替原来的主语出现。

最后的输出有两部分，如图 6-69 所示，就是该单元生成的 h_t，一个输出到同一层的下一个单元，另一个输出到下一层的单元，其表达式为：

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o);$$
$$h_t = o_t \cdot \tanh(\boldsymbol{C}_t).$$

其实可以看出，要输出的向量 $\boldsymbol{C}_t$ 又经过了一个忘记门 o_t 的乘积来输出，成为 h_t。在语言模型中，这种现象会影响前后词之间词形的相关性。例如，在英语环境中，前面输入的是一个代词或者名词，后面跟随的动词会学到是否使用“单三形式”，或者根据前面输入的名词数量来决定输出的代词是单数形式还是复数形式。

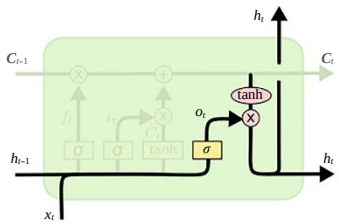


图 6-69　LSTM 单元（5）

在 NLP 领域，LSTM 的应用比较成熟和广泛。在本节中虽然出现了很多公式，但读者朋友也不用紧张。对这些公式，不需要强行记忆（无论是在就业面试中，还是在实际工作中）。我们在这里只是了解一下 LSTM 的大致工作原理，在实际工程应用中，我们还是会用 PyTorch 封装的库来实现。在深度强化学习中，会有个别模型使用 LSTM 单元，但是从比例上看，数量非常少。

对于一个 LSTM 单元内部的众多参数来说，我们只能使用，没有什么调整的余地。训练出来的 LSTM 网络，看上去也几乎没办法做任何解释。所以，了解到这里就可以了。我们只要知道，一个以时间序列输入的 Sequence，如果其前后有联系，就可以根据上文对下文进行提示或限制，那么，使用 LSTM 是可以对它进行学习的。也可以说，通过 LSTM 的时序性信息的输入和时序性信息的输出，让 LSTM 网络去记忆时序之间的概率关系。

6.11.2　应用案例

工业界对于 RNN 的应用案例，我个人见得并不多，至少没有以卷积层和全连接层组成的模型网络多。然而，在学术界，RNN 的应用还真不少，我们一起来看。

LSTM 组成的网络，可以作为分类工具，可以作为有限状态机（控制程序），可以用于实现翻译器，可以用于实现聊天机器人等看上去很酷的东西……如图 6-70 所示，是对一个 RNN 可以做的事情的“可视化”归纳，大致是“一到一”“一到多”“多到一”“多到多”的映射。

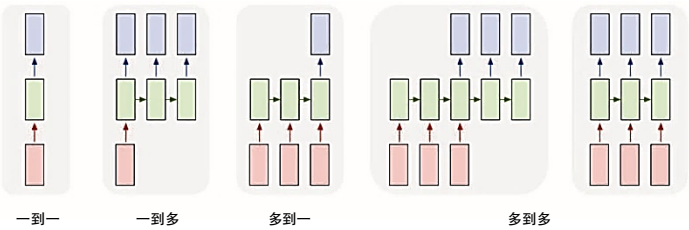


图 6-70　LSTM 网络的多种形式

“一到一”很简单，普通函数能做的事情，RNN 肯定能做（即输入一个一维或多维的向量，输出一个一维或多维的向量）。

“一到多”是一种单向量输入、多向量分时序输出的场景，例如大名鼎鼎的 Neuraltalk2（https://github.com/karpathy/neuraltalk2），如图 6-71 所示。

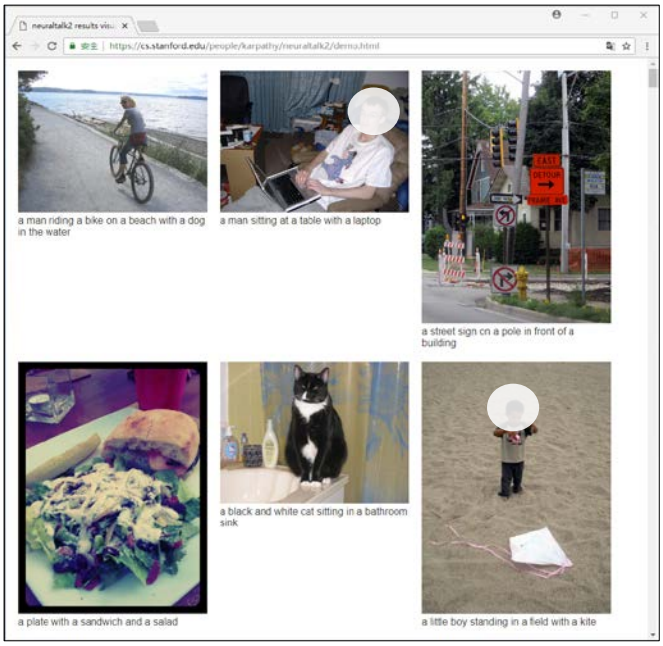


图 6-71　Neuraltalk2

Neuraltalk2 可以根据输入的图片生成一个描述图片内容的短句。我们来看页面上提供的这几个，识别效果还不错。第一行中间这幅图片，是一个男人在玩笔记本电脑，给出的英文说明是“a man sitting at a table with a laptop”（一个男人正带着笔记本坐在桌子旁），还是很贴切的。而第一行左边这幅图片，就有点问题了，英文说明是“a man riding a bike on a beach with a dog in the water”（一个男人正在海滩上骑自行车，还有一条狗在水里），如果由人来识别，就可以看出，这个男人确实是在水边的公路上骑自行车，可是狗在哪儿？反正我是没找着。每到这种时候，我就会开玩笑说：“不知道神经网络发了什么‘神经’……”

Neuraltalk2 是由两部分网络连接而成的。如图 6-72 所示，网络的前半部分是一个卷积神经网络（Vision Deep CNN 部分）。在图片分类任务上进行训练，把通过训练得到的权重保存下来，这个网络就拥有了较强的分类能力。把网络的最后一层去掉，不让它完成分类任务，留下从第一层到倒数第二层之间的部分。这样把网络“锯开”后，当一幅图片从网络的前端输入，在最后一层看到的就不再是分类概率了，而是提取出来的一些低维连续空间上的图片特征，也就是 Feature Map 记载的东西。尽管这些特征都是用矩阵来表示的，人是读不懂的，但是，特征实实在在地被提取出来了，不然也不可能在最后实现分类。

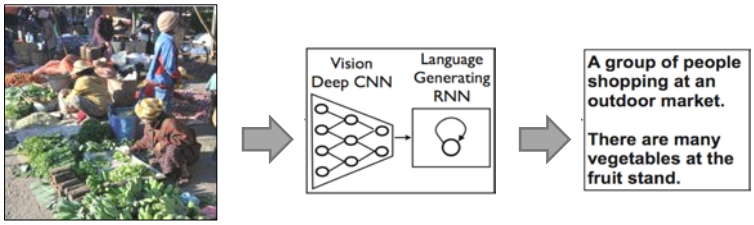


图 6-72　Neuraltalk2 模型结构示意

接下来，把表示特征的矩阵作为后半部分网络的输入。后半部分网络就是 RNN（Language Generating RNN 部分），可以用 LSTM 网络实现。图片上的像素也好，录制的一段语音也罢，只要以时序形式作为输入，就是一个根据前面的输入内容来推断后面的输出内容的过程（因为这些信息之间有依赖性），因此，可以用 LSTM 网络实现预测。

“多到一”可用于实现视频主题分类。例如，C3D: Generic Features for Video Analysis 项目（http://vlg.cs.dartmouth.edu/c3d/）就可以实现视频主题的识别。这个项目可以比较好地对体育竞技类内容进行分类识别，如图 6-73 所示。

图 6-73　C3D 项目效果图

“多到多”的用例是比较多的，而且是研究热度非常高的一个领域。

例如，聊天机器人可以在客服、问讯系统等场景中落地，从而减少人工方面的投入。再如，在谷歌和百度这样的搜索引擎中，自动翻译机器人已经有了很好的落地应用，在一些场景中，译文的准确程度已经可以与水平较高的翻译人员相比了。如图 6-74 所示，除了措辞有失文采和感情外，译文的通顺程度基本达到了人类可以正常阅读的标准，就连复杂的从句也翻译得非常好。这种自动翻译机器人比任何一种传统的单词查询工具都好用太多了，给需要跨语言阅读的人带来了极大的便利。

一些描述视频段信息的 RNN 项目（例如 https://github.com/samim23/NeuralTalkAnimator）也属于“多到多”的应用场景。例如，在一段视频中，RNN 能够正确地识别出图像中有一群人站在绿色的草坪上。

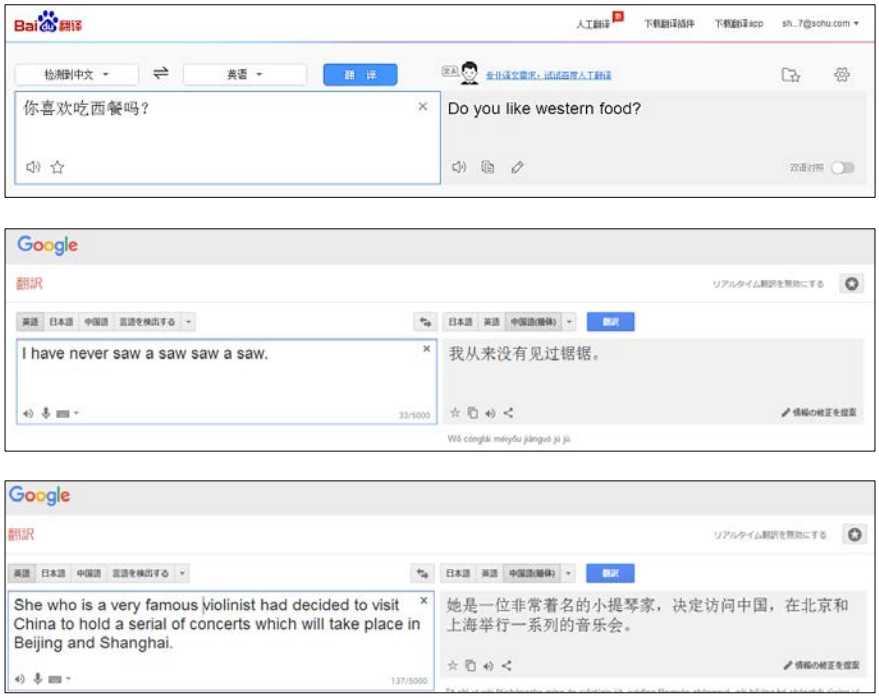


图 6-74　自动翻译项目效果图

越来越多的人工智能项目已经落地，越来越多的技术正在发展。也难怪总有人说，“有些行业快要集体失业了”。对于这样的论调，我觉得，这个行业的发展趋势是典型的“劣币驱逐良币”，越来越多人工智能项目的落地，使得大量自动化处理数据工作能够以低廉的成本完成，所以，我认为，要不了多久低端的服务市场就不会再有人做了，因为人的工时价格再低，也不可能比机器的工时价格低。不要觉得人工智能应用的研发成本很高，还是人的工时价格比较便宜，我们要明白：有些东西一旦研究出来，其产出是人工望尘莫及的。机器逐步替代人的大趋势是无法扭转的，我们现在要像第一次工业革命时期的手工匠人一样，要么去做那些极为精巧、无法用机器做出来的东西，要么去研究如何制造或者操控机器，否则，就真的要落后于时代了。

6.11.3　代码解读

看了这么多应用案例后，我们还是来看看手边的代码。

这是一段用 LSTM 网络识别 MNIST 手写数字的代码。在 ~/DRLPytorch/recurrent_neural_network 目录中，打开 main.py 文件，内容如下。

```
1 import torch
2 import torch.nn as nn
3 import torchvision.datasets as dsets
```

```
 4 import torchvision.transforms as transforms
 5 from torch.autograd import Variable
 6
 7
 8 # Hyper Parameters
 9 sequence_length = 28
10 input_size = 28
11 hidden_size = 32
12 num_layers = 2
13 num_classes = 10
14 batch_size = 100
15 num_epochs = 2
16 learning_rate = 0.01
17
18 # MNIST Dataset
19 train_dataset = dsets.MNIST(root='./data/',
20                             train=True,
21                             transform=transforms.ToTensor(),
22                             download=True)
23
24 test_dataset = dsets.MNIST(root='./data/',
25                            train=False,
26                            transform=transforms.ToTensor())
27
28 # Data Loader (Input Pipeline)
29 train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
30                                            batch_size=batch_size,
31                                            shuffle=True)
32
33 test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
34                                           batch_size=batch_size,
35                                           shuffle=False)
36
37 # RNN Model (Many-to-One)
38 class RNN(nn.Module):
39     def __init__(self, input_size, hidden_size, num_layers, num_classes):
40         super(RNN, self).__init__()
41         self.hidden_size = hidden_size
42         self.num_layers = num_layers
43         self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=
True)
44         self.fc = nn.Linear(hidden_size, num_classes)
45
46     def forward(self, x):
47         # Set initial states
48         h0 = Variable(torch.zeros(self.num_layers, x.size(0), self.hidden_size))
49         c0 = Variable(torch.zeros(self.num_layers, x.size(0), self.hidden_size))
50
51         # Forward propagate RNN
```

```
 52        out, _ = self.lstm(x, (h0, c0))
 53
 54        # Decode hidden state of last time step
 55        out = self.fc(out[:, -1, :])
 56        return out
 57
 58 rnn = RNN(input_size, hidden_size, num_layers, num_classes)
 59
 60
 61 # Loss and Optimizer
 62 criterion = nn.CrossEntropyLoss()
 63 optimizer = torch.optim.Adam(rnn.parameters(), lr=learning_rate)
 64
 65 # Train the Model
 66 for epoch in range(num_epochs):
 67     for i, (images, labels) in enumerate(train_loader):
 68         images = Variable(images.view(-1, sequence_length, input_size))
 69         labels = Variable(labels)
 70
 71         # Forward + Backward + Optimize
 72         optimizer.zero_grad()
 73         outputs = rnn(images)
 74         loss = criterion(outputs, labels)
 75         loss.backward()
 76         optimizer.step()
 77
 78         if (i+1) % 100 == 0:
 79             print ('Epoch [%d/%d], Step [%d/%d], Loss: %.4f'
 80                    %(epoch+1, num_epochs, i+1, len(train_dataset)//batch_size,
loss.data[0]))
 81
 82 # Test the Model
 83 correct = 0
 84 total = 0
 85 for images, labels in test_loader:
 86     images = Variable(images.view(-1, sequence_length, input_size))
 87     outputs = rnn(images)
 88     _, predicted = torch.max(outputs.data, 1)
 89     total += labels.size(0)
 90     correct += (predicted == labels).sum()
 91
 92 print('Test Accuracy of the model on the 10000 test images: %d %%' % (100 *
correct / total))
 93
 94 # Save the Model
 95 torch.save(rnn.state_dict(), 'rnn.pkl')
```

- 1 ~ 35 行：这部分我们看过三次了，不再多说。只需要注意，9 ~ 16 行中有一些超参数（会在 6.12.4 节介绍）。
- 37 ~ 63 行：网络的核心部分，定义了网络结构和损失函数。43 行定义了 LSTM 网络层，包括 `input_size`（输入的尺寸）、`hidden_size`（LSTM 单元隐藏层的数量）、`num_layers`（LSTM 单元的层数）和 `batch_first`（赋值为 `True`，表示输入的 `Tensor` 的第一个维度是 `batch` 信息）。

对一个样本来说，这是一个二维的数据，如图 6-75 所示。我们将输入的二维张量的两个维度分别称作 x 和 y。

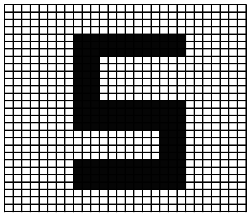


图 6-75　MNIST 数据 28×28 格式示意图

在这个例子中，`batch_size` 的值为 100，表示每次会传入一个三维的张量，它有 100×28×28 这么大。既然它是由一个“长方体”传入的，那么它是“躺着”的、“站着”的还是其他姿势，需要跟 LSTM 网络交待清楚（这会影响 LSTM 网络的处理结果）。参数既可以是 [100,28,28]，也可以是 [28,28,100]。这个参数用于声明第一个维度指的是 `batch` 还是时序信息。如果 `batch_first` 为 `True`，就是 [100,28,28]，x 维相当于一个个样本，y 和 z 则分别是原来单个样本中的 x 和 y（如图 6-76 所示）。`hidden_size` 和 `num_layer` 用于声明这个 LSTM 层的尺寸。`hidden_size` 为 128，`num_layers` 为 2。在这个 LSTM 层中，Cell 也是以这种“叠罗汉”的方式实现的。

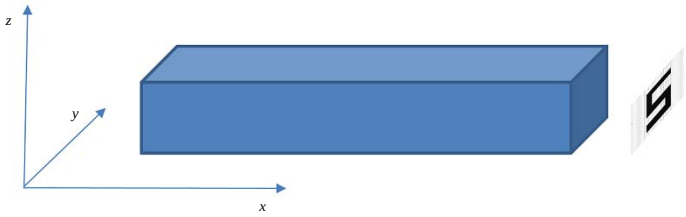


图 6-76　批量输入的字符图片

在 6.11.2 节中介绍过，LSTM 由于自身的特殊性质，可以实现“一到一”“一到多”“多到一”“多到多”的映射拟合。这个例子是“多到一”的映射，输入是 28 行，每行是一个 28 维的向量，输出是一个独热向量的分类标签，其网络结构如图 6-77 所示。

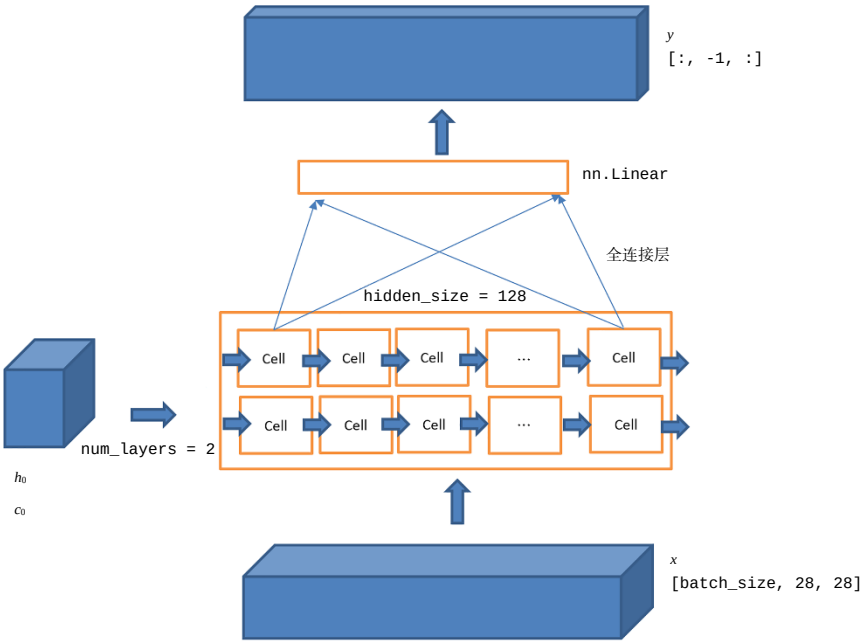


图 6-77　本实验的 LSTM 示意图

48 ~ 49 行初始化左边第一次输入的值，输入之后，h_1/c_1、c_2/c_2……会顺着网络向右传输（在这里都初始化为 0）。55 行定义了网络的输出。在 LSTM 网络输出的内容中，取第一个维度的全部、第二个维度的最后一行和第三个维度的全部。

所谓最后一行，其实是将 28 × 28 的矩阵从上到下扫描后的最后一次输出，可以理解为记住了所有有用的数据关系（如图 6-78 所示）。就像一个人说了 28 句话，在最后一句中才说明“这是数字 5”。这就是输出逻辑的定义，每一句话就像一行数据（28 维）。

```
out = self.fc(out[:, -1, :])
```

在这里，`out[:, -1, :]` 是指把输出的第二个维度的最后一个切片拿出来进行全连接的输入。第二个维度，就是沿着 `sequence_length` 往下走得到的最后一个 Sequence，也就是一幅图片的“28 行”中的最后一行。而通过 `out[:, -1, :]`“切”出来的是整个“长方体”中的一个片。这个片是一个二维张量，第一个维度是 `batch_size`，第二个维度是 `input_size`，尺寸为 [100,28]。

损失函数仍然是交叉熵损失函数（`CrossEntorpy()`）。也就是说，在将这 100 个样本的最后一个输入后，将输出的 28 维与一个 10 维的输出进行全连接操作，再进行交叉熵运算。

“多到一”的定义其实并不复杂，关键是要把 `size` 的值弄对了。至于其他参数，我们几乎没有修改的权限。

- 66 ~ 98 行：训练过程，其中 82 ~ 95 行是验证并存储模型的部分，就不再赘述了。

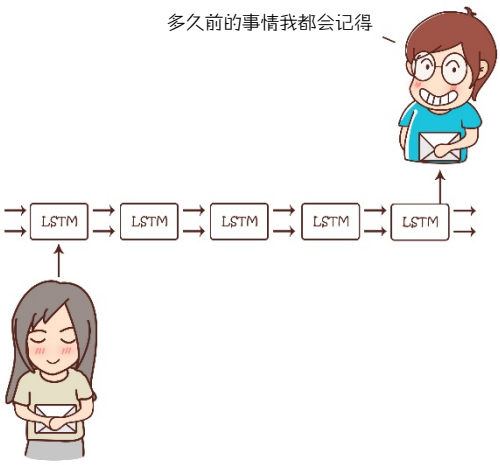


图 6-78　LSTM 的“记忆”

在这个实验中，可以尝试修改 `hidden_size`，让 Cell 的数量少一些（例如修改为 64），这样训练的速度也能快一些。在 4 核的 CPU 上进行训练，大约需要 2 分钟，正确率为 96%～97%。

6.12　其他注意事项

按说到这里，深度学习的“门”我们应该算是“入”了。是的，确实“入”了，不过“门槛”在“门”里面——有点像相声。没错，艰深的东西还有很多，只不过我们技巧性地进行了躲避。

尽管在研究足够深入之前，你不会有太深的感觉，不过，仍然有一些知识点需要补充学习，以免在实际工程中被一些问题困扰。

6.12.1　并行计算

并行计算应该是所有用于加速训练的方案中最招人喜欢的了。为什么你会觉得深度学习“学”起来速度非常慢呢？究竟慢在哪儿呢？说到底，还是计算资源的问题所致——深度学习的训练过程，对计算资源的需求是非常大的。

不管是全连接网络，还是卷积神经网络、循环神经网络，在做正向传播的时候，都需要进行大量的乘法运算——准确地说，是矩阵运算。这个问题是由深度学习本身的模型定义造成的。网络越深，参数越多，“吃”计算资源的现象就越明显。但是，为了提高网络的拟合能力，在很多时候不得不确保网络拥有足够的层和参数。于是，就只剩下一条路——努力提高计算能力了。

CPU 的计算能力是非常有限的。以市面上的个人计算机为例，如表 6-5 所示，2018 年，大部分主流的 CPU 在 10 个内核左右（甚至达不到），高端的有 18 核 36 线程的 i9 7980XE（在写作本书时，其市场零售价为 15000 元左右）。

表 6–5　2018 年主流个人计算机的 CPU 参数

型　　号	主频（GHz）	核心数量	线程数量
i5 7640X	4.0	4	4
i7 7740X	4.3	4	8
i7 7800X	3.5	6	12
i7 7820X	3.6	8	16
i9 7960X	2.8	16	32
i9 7980XE	2.6	18	36

而对训练模型，哪怕只用一个 VGG-16[①]网络在 MS COCO[②]上进行图片分类训练，恐怕都要花上两个星期不止——这简直糟透了。

比较成熟的是英伟达（NVIDIA）提供的解决方案，如表 6-6 所示。这是一种在 GPU 中使用的并行计算单元 CUDA（Compute Unified Device Architecture）。在一块具有 CUDA 功能的显卡中，有数以千计的 CUDA 内核（相当于 CPU 内核）。

表 6–6　NVIDIA 的 GPU 参数

型　　号	CUDA 核心数量	显存（GB）
GTX 1060	1280	6
GTX 1070	1920	8
GTX 1080	2560	8
GTX 1080TI	3584	11
TITAN XP	3840	12
P100	3584	16
V100	5120（640 个 Tensor Core）	16

一般来说，CUDA 的核心数量越多，其并行计算能力就越强，在同等情况下训练速度就越快。因为在训练时需要将样本数据复制到显存中，如果样本集很大，会直接“撑爆”显存，导致无法训练，所以，显存大的显卡对于训练大样本集来说是有好处的。除了这些指标，还需要参考显存位宽、GPU 主频等影响运算速度的因素，即便是入门级，也要使用 GTX 1070 这样的显卡才行。显存过小容易给训练带来限制，随随便便就会出现的 OOM[③]错误使我们不得不减小 `batch_size` 的值来适应狭小的显存空间。

V100 拥有 5120 个 CUDA 内核，并增加了 640 个 Tensor Core——这是一种为深度学习量身定制的计算单元。V100 的价格不算便宜，在写作本书时，其价格为 47000 元左右。2018 年，在我生

① 一种神经网络模型，由牛津大学计算机视觉组发明及提出。
② 微软提供的图片分类公开数据集。
③ Out of Memory，译为“内存溢出”。

活的城市珠海，这笔钱能买 2 平方米海景房了。当然，它的表现提升，比起报价仅为 3000 元左右的 GTX 1070，或许到不了 10 倍（在小模型的训练上提升更不明显）。如果是个人入门，GTX 1070 已经能够满足需求了——不管是入门学习，还是运行一些计算消耗比较大的图形处理模型，都应该够用了。我们也可以选择性价比较高的 GTX 1080Ti（5500 元左右一块），它也是很多预算有限的公司的首选。

CUDA 作为 NVIDIA 的平台级产品架构，不仅有丰富的开发文档，还支持复杂的并行计算编程。不过，对我们这些应用层面的 PyTorch 使用者来说，不必了解这些东西，因为在底层驱动层面，不论是 PyTorch、TensorFlow、MXNet，还是其他框架，NVIDIA 都为我们提供了透明化的解决方案，我们只需要指定训练或分类过程中使用的 GPU 就可以了。

对于多 GPU 环境，例如 4 路或者 8 路 GPU 的工作站（服务器），有很多 NVIDIA 授权代理商在做相关的工程服务，包括安装显卡、CPU、内存、风墙，在必要的时候可以安装液冷系统（如图 6-79 所示）。大家可以访问 http://www.nvidia.cn/object/where-to-buy-tesla-cn.html，查看 NVIDIA 官方认证的合作公司，选择合适的供货商。不限于大型机构（戴尔、惠普、华为等），小型机构也可以获得一些价廉物美的零售方案。

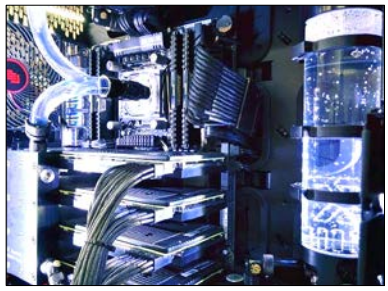


图 6-79　NVIDIA 显卡和水冷降温设备

如果对训练效率的要求比较高，还是建议选择 NVIDIA 原厂的产品（如图 6-80 所示），或者咨询 NVIDIA 授权代理商。这些产品通常有比较高的质量，NVIDIA 也会在系统层面对硬件及深度学习框架进行优化——据说最多能提高 30% 的效率。

图 6-80　NVIDIA DGX-1 工作站

作为企业技术人员或者研究人员，我们应该把更多的精力放在对业务的研究和改进上，像硬件解决方案这样的工程级问题，最好交给专业的人去做。

CUDA 提高深度学习训练速度的原理是，发挥并行计算的优势，主要对样本的正向传播过程进行计算。不过，需要格外注意的是，这种效率的增益在离线学习（批处理）中比较有优势，例如典型的监督学习（批处理离线学习），而在一边收集样本、一边做估值调整、一边做梯度下降拟合的场景中，优势就不明显了。

在离线学习场景中，一个 Batch 里面可以放置多个样本。在训练中，只需要完成一项任务，就是尽快通过 CUDA 的并行计算（如图 6-81 所示）实现正向传播，并通过反向传播误差把损失函数的极值点找到。在强化学习中，大都会面临冷启动的问题。由于此时根本没有样本可以学习，就要使用大量 CPU 资源去试探，收集大量 $S \to A \to R \to S'$ 的数据。此外，在 TD 算法中，还要用贝尔曼方程来逐步更新 $Q(s,a)$，“制作”并逐步调整“标签”。要知道，如果 $Q(s,a)$ 不准确，即便拟合出深度神经网络，也是没有意义的。这个调整和回溯的过程不是梯度下降过程，PyTorch 或者其他任何一个神经网络框架都帮不上忙。

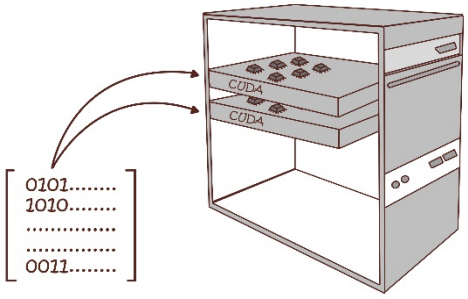


图 6-81　CUDA 的并行计算

因此，在实际的工程实现中，针对强化学习的训练，矛盾可能更多地体现在使用大量的 CPU 提高样本收集的并行性上，而由梯度下降产生的计算矛盾却不那么明显（当然，这还要取决于场景的复杂程度和神经网络的复杂程度）。有一个原则是确定的：神经网络的复杂程度越高，层和待定系数的数量越多，训练所需的 GPU 计算资源就越多。

在离线学习场景中，你可能还会看到多个 GPU 一起训练的情况。如果有 4 个 GPU，它们都支持 CUDA 且驱动都安装正常的话，在使用 PyTorch 时，需要显式地告诉模型：要使用多个 GPU 来进行训练（要用 DataParallel 这个类实现）。4 个 GPU 怎么使用呢？能不能提高训练速度呢？可以肯定的是，在离线学习场景中，尽管 4 个 GPU 肯定比 1 个 GPU 的计算速度快，但是效率达不到 1 个 GPU 的 4 倍。根据我的经验，能将训练效率提高为原来的 2 倍就不错了，原因如下。

一是样本进入显卡的 I/O 时间是不能忽略的。每个 Batch 操作都需要几十毫秒，性能高的机器耗时可能会短一些，可以达到十几毫秒或者更短。然而，这个时间对于 GPU 计算来说已经很长了，I/O 对 GPU 计算产生的“空档”已经让计算效率大打折扣了。

二是多 GPU 加速的原理是计算平均梯度。以本节前面的例子来说，原来一个 GPU 一次可以

处理 batch_size 为 100 的计算规模，换成 4 个 GPU 之后，这 4 个 GPU 会分别同时处理 100 个样本，分别根据 100 个样本的正向传播得到 4 个不同的误差值，然后根据这 4 个不同的误差值算出 4 个不同的梯度，再把这 4 个梯度归拢到其中一个 GPU 上，求出平均梯度，并在这个 GPU 上进行更新，把更新后的模型分发到 4 个 GPU 上——如此反复。所以，在使用多个 GPU 时，你可能会看到其中的一个使用率特别高，另外几个使用率要低一些的情况——是的，另外几个 GPU 在趁着“主”GPU 更新网络的时候“打酱油”，而“主”GPU 也没有提高更新权重的效率。

从这个角度看，训练速度不会随 GPU 数量的增加呈线性增长。

6.12.2 梯度消失和梯度爆炸

虽然在前面囫囵吞枣式的学习中，我们确实已经了解了如何使用 PyTorch 来搭建简单的全连接网络、卷积神经网络和循环神经网络，但实现层面的原理我们并没有提及。一是对于快速入门 PyTorch、搭建一个可训练的网络来说，掌握基本知识就够了；二是在一开始就介绍太多的底层实现，会让读者感觉摸不着头脑（就像一个人本来打算学怎么用砖盖房子，结果一上来学的是怎么烧砖一样）。

要讲解梯度消失和梯度爆炸，就要先介绍梯度下降，这是计算机在帮我们进行训练时必须要使用的一种计算方法。

1. 梯度下降

梯度下降（Gradient Descent），就是前面多次提到的通过求偏导数来求出更新量，然后让待定系数沿着梯度下降的方向更新的过程。

2. 随机梯度下降

除了梯度下降，还有一个术语叫作随机梯度下降（Stochastic Gradient Descent，SGD）。随机梯度下降在前面虽未做详细讨论，但在本书中不时会提到。它和梯度下降有什么区别？我们会使用它吗？其实，我们一直在使用它。

在介绍梯度下降的时候，我们是按照损失函数的定义

$$\text{Loss} = \frac{1}{n}\sum_{i=1}^{n}(f(x_i|\theta) - y_i)^2$$

来讨论的（n 是全样本的数量）。例如，在 MNIST 数据集中有 50000 个训练样本，那么 n 的值就是 50000。问题也就来了：将这么多数据一次性载入内存是一件非常可怕的事，尤其在一些大型网络的目标识别任务中，对于内存或者显存的消耗动辄数十吉字节（GB）。所以，它在工程应用上有着天然的障碍。

我们也知道，在统计学中有一个概念叫作抽样（Sampling），就是随机在一个大的样本库中找一小部分来代表这些样本的特征，如图 6-82 所示。随机梯度下降就利用了抽样的原理。例如，有

50000 个训练数据样本，每次只挑选其中 100 个作为一个 Batch，下一次再换 100 个，这样一次次进行更新，整个网络最终接收的样本数将趋近于全样本。如果在程序中强制规定已经被抽取的样本不再放入样本库，那么，在一个 Epoch 结束的时候，每个样本都会通过网络一次。

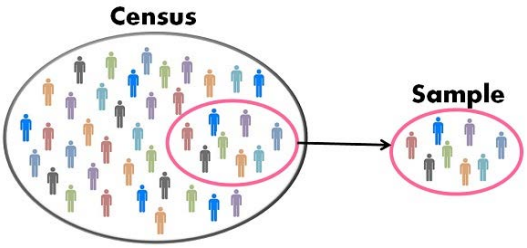


图 6-82　抽样

这种每次挑选一部分样本作为代表进行训练而产生的梯度下降就是随机梯度下降。在工程应用中，一直在使用这种方法。

随机梯度下降方法的优点很明显，就是减少每一次梯度更新过程的计算量，使得很多样本量极大的训练过程变得可行。然而，这种方法的缺点也是很明显的，那就是由于每次选择的都是部分样本，部分样本和全样本之间的关系虽然具有代表性，但分布特征很难完全吻合。如图 6-83 所示，虚线箭头表示随机梯度下降，实线箭头表示确定方向的梯度下降。可以看出，在做随机梯度下降的梯度更新时，方向不定，路径显得比较曲折，而对于确定方向的梯度下降，更新的方向是非常明确的。

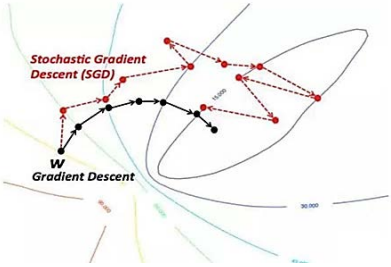


图 6-83　梯度下降的方向

随机梯度下降方法瑕不掩瑜，大家大胆地使用吧。

3. 梯度消失

梯度消失（Vanishing Gradient）是在早期的神经网络中比较常见的问题。我们还是来看一个具体的例子。

最远一侧的待定系数偏导数为

$$\frac{\partial \text{Loss}}{\partial w_1}=\frac{\partial \text{Loss}}{\partial z_2}\frac{\partial z_2}{\partial x_2}\frac{\partial x_2}{\partial z_1}\frac{\partial z_1}{\partial w_1},$$

还是一个由两个节点首尾相接组成的神经网络（应该说，连“线”都算不上）。

如图 6-84 所示，对位于网络前部的 w_1 的更新，需要计算损失函数对 w_1 的偏导数。

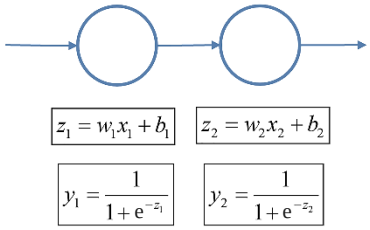


图 6-84　“串联”的网络

可以发现，$\frac{\partial \text{Loss}}{\partial z_2}$ 和 $\frac{\partial x_2}{\partial z_1}$ 这两项实际上是在寻找 Sigmoid 函数图象上的斜率。Sigmoid 函数我们肯定不陌生了，前面已经提过很多次了，如图 6-85 所示。

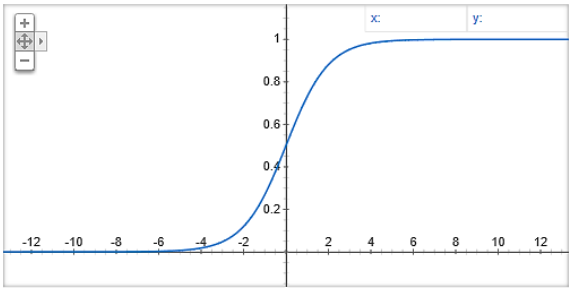


图 6-85　Sigmoid 函数

这个函数最大的特点就是：我们可以用肉眼看到，自变量在大于 4 和小于 –4 的两段中几乎呈水平状态，即导数的值接近 0。这就比较麻烦了，如果 $\frac{\partial \text{Loss}}{\partial z_2}$ 或 $\frac{\partial x_2}{\partial z_1}$ 中的任何一个处于这两段中，会让整个导数的值接近 0，这样，误差在向前传递的过程中会导致网络前端的 w 几乎没有变化。计算出来的导数值实在太小了，乘以一个 λ 就变成了一个微乎其微的值，而且，网络层级越多，在越靠前的隐藏层中这种情况就越严重（w 的变化速度就越慢）。也就是说，在这一层没学到什么东西。这就是我们所说的梯度消失问题，也叫作梯度弥散问题（如图 6-86 所示）。

改进的思路，无非就是消除这种在链式法则中发生的连乘式中的每一项的绝对值小于 1 的情况。因为在绝对值小于 1 时，网络的层级越多，就意味着有越多的绝对值小于 1 的数字相乘——越乘越小。网络层数稍微多一点，在越靠前的层级上，w 移动的速度就越慢，学习率就越低。也就是说，在这个 2 层的网络中，我们要解决这样的问题：

$$\frac{\mathrm{d}z_2}{\mathrm{d}x_2} = w_2;$$

$$\frac{\mathrm{d}x_2}{\mathrm{d}z_1} = \sigma'.$$

σ' 的意思是对 Sigmoid 函数求导，即

$$\sigma(x)' = \sigma(x)\big(1-\sigma(x)\big),$$

也可以写成

$$\sigma(x)' = \frac{1}{1+\mathrm{e}^{-x}} \cdot \Big(1 - \frac{1}{1+\mathrm{e}^{-x}}\Big).$$

图 6-86　对梯度消失的理解

那么，要使得 $|w\sigma'| \geqslant 1$，该怎么办呢？可能很容易想到两种思路。

（1）初始化一个合适的 w

如果能够把 w 的初始值设置得大一些，是不是就能解决这个问题了呢？把 w 的初始值设置为 10，在 Sigmoid 函数的值为 0 的地方，导数的绝对值是 $|\sigma'(0)| = \frac{1}{4}$，当 $|w| = 10$ 时，$|w\sigma'| = 2.5$。这个值确实够大。但是，大就一定好吗？未必。

同样是链式法则的连乘关系，原来是因为导数太小导致网络前端的 w 变化速度太慢。这么一改，反过来了，网络前端的变化率太大（即一次的变化量非常大）。就拿前面这个例子来说，如果有 10 层，10 个 2.5 相乘约等于 9536.7。过犹不及——就算是一个很小的 η，也能一下子从“地下”挪到“天上”。这种现象叫作梯度爆炸（Gradient Exploding），也叫作梯度膨胀。同时，w 在移动的过程中也一直在变化，怎么可能保证这种效果的持续性呢？

（2）选择一个合适的激励函数

解决这个问题最为成熟的方法，就是使用导数值比较合适的激励函数，例如 ReLU 函数（如

图 6-87 所示)。ReLU 函数在 6.4.3 节曾经介绍过，现在我们可以仔细讨论一下了。

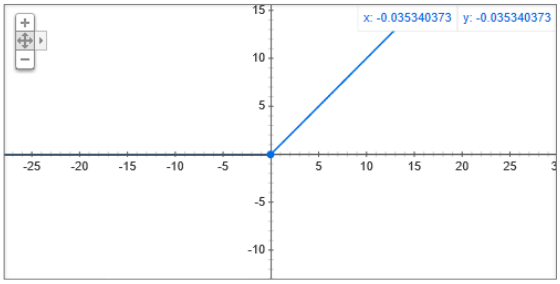


图 6-87　ReLU 激励函数

“ReLU”一般读作“热鲁”，其全名是“Rectified Linear Units”，常被翻译成“线性修正单元激活函数”。

ReLU 函数的形式为 $y = \max(x, 0)$。在这个函数的图象中，原点左侧部分斜率为 0，右侧则是一条斜率为 1 的直线。函数在 x 大于 0 的部分呈现出线性的特点，在小于 0 的时候则是一条直线。这个锋利的弯折提供了良好的非线性特点。尤其要注意第一象限，导数值恒为 1，这是它的“激活”状态。而当 x 小于 0 时，导数值恒为 0，这是它的“非激活”状态。所以，它的如下两个优点是显而易见的。

- 在第一象限中不会有明显的梯度消失问题。因为导数值恒为 1，w 在初始化的时候是有大有小的，所以在连乘的时候不会轻易出现很小或者很大的值。
- 因为导数值为 1，所以求它的导数要比求 Sigmoid 函数的导数的代价小一些。这里说的代价主要是时间代价。

因此，工程人员在网络中喜欢大量使用 ReLU 函数。

4. 其他优化器

关于利用梯度下降的方法找到损失函数的极小值的原理，我们已经了解了。然而，这里仍然有一个比较严重的问题。

如果损失函数不是纯粹的凸函数，不能通过一个方向持续更新，进而走到最低点，该怎么办呢？换句话说，由于损失函数的图象就像一只高维且有褶皱的碗，这会不会让更新点陷在某个褶皱里爬不出来？此外，可以证明，如果一个网络中的每一层都是线性关系，而没有非线性的激励函数，平方损失函数就是标准的凸函数。也就是说，除此之外，就是一只满是褶皱的碗。又该怎么办呢？在最优化问题的研究领域，数据科学家们找到了很多方法。

在 PyTorch 中，除了 GD（SGD），还有 Momentum、RMSprop、Adam 算法，它们是一代一代升级改进而来的。如果非要定性地描述——GD（SGD）的收敛速度最慢，其次是 Momentum，RMSprop 和 Adam 的差距不明显，但比起前两种算法，收敛速度要明显快一些。它们所对应的类

包括 `torch.optim.SGD`、`torch.optim.RMSprop`、`torch.optim.Adam`（因为 Momentum 是 SGD 的特殊形式，所以没有单独实现它的类）。

在这里，作为知识点的补充，我们简要介绍一下 Momentum 算法的实现原理。

Momentum 算法，也叫作使用动量的随机梯度下降算法。SGD-Momentum 的伪代码如下。

设置学习率 ε，动量参数 α
设置初始参数 θ，初始速度 v
循环：
　从训练集中抽取 m 个样本，每个样本称为 x_i、y_i
　计算梯度估计：$g \leftarrow \frac{1}{m}\nabla_\theta \sum_i L(f(x_i;\theta), y_i)$
　计算速度更新：$v \leftarrow \alpha v - \varepsilon g$
　应用更新：$\theta \leftarrow \theta + v$

这段伪代码有点太“伪”了，还是解释一下吧。SGD 的过程该怎么做还怎么做，但每次更新 θ 时，不是直接使用求解出来的梯度 g，而是先减 εg 再加 αv，用这个结果进行更新。最后的 $\theta \leftarrow \theta + v$，表示真实的更新梯度值是 v。而 αv 中的 α 作为一个衰减系数，记录（或者说保持）了上一次梯度的更新方向，让它“带着一股冲劲儿往前跑”。同时，用减 εg 的方法来记录这次的一个 Batch 样本的梯度值，因此，减 εg 就相当于减 ηg（也就是 SGD 算法中的梯度计算方法）。

采用这种方式，就可以让更新点有比较大的概率“离开”当前陷入的褶皱，继续往更低的地方“跑”。

6.12.3　归一化

归一化问题（Normalization）是几乎所有机器学习算法在开始训练之前都要考虑的问题。所谓归一化问题，就是为了克服这样一种现象而采取的应对方法。是什么呢？我们看一个例子。

假设我们要计算中国人和日本人的平均收入，比较差距，应该怎么做？从统计学的角度，肯定要在中国人和日本人中随机抽取一部分样本（例如各 1000 人），分别计算他们的平均年收入。相对来看，这种方法的科学性比较强，也比较有说服力。

但是，你会发现，计算后有可能得到这样的结果：

```
中国人平均年收入：55,000；日本人平均年收入：2,600,000
```

这两个数字的差距实在是太大了——超过了 46 倍。如果这时得到的结论是日本人的平均收入是中国人的 47 倍，未免太过荒唐了。

要知道，在获取样本的时候，都会以当地的标准货币为计算单位。如图 6-88 所示，按照写作本书时人民币对日元的汇率（约 1 比 16.28）换算，中国人的平均年收入可以大致核算为 89.57 万日元，也就是说，日本人的平均年收入大约是中国人平均年收入的 2.90 倍——靠谱多了。

图 6-88　人民币与日元的兑换

对有货币汇兑换常识的人来说，换算过程肯定是不可避免的。然而，在计算机系统中，数字是没有量纲和单位的，只有一个具体的浮点数或者定点数。这就麻烦了，在机器学习中，由于计数单位的影响，分布范围较大的值和分布范围小的值会在训练过程中表现出不同的影响能力，其后果主要是使得模型对某些值过于敏感或者过于不敏感——这种情况其实是我们不愿看到的“不公平”的情况（如图 6-89 所示）。解决方法也是有的，那就是使用归一化这样一个操作过程，把数据的分布压缩或限定在一个比例比较协调的范围之内。

图 6-89　归一化的重要性

常见的归一化方法，主要包括线性函数归一化（Min-Max Scaling）、0 均值标准化（Z-Score Standardization）等，其目的都是让各维度的数据分布经过“伸缩”，投射到一个合理的范围内。

1. 线性函数归一化

线性函数归一化是一种常用的归一化方式，其表达式为

$$X_{\text{norm}} = \frac{X - X_{\min}}{X\min_{\max}}.$$

每个数值经过这样的投射，变成了 0 和 1 之间的数值。这个值表示在整个样本中该维度所处的位置（比例）。

经过线性函数归一化处理，数据的分布会大致呈现如图 6-90 所示的效果。左边这幅图是原始数据。中间这幅图经过中心化处理，也就是分别用两个维度的值减去各自的平均值，得到一些有正有负的值且 0 在中心位置。右边这幅图是处理的结果，很明显，数据分布的矩形轮廓不再是一个狭长的形状，而是趋近于正方形了。

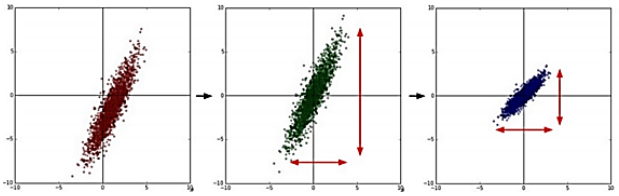


图 6-90　线性函数归一化

2. 0 均值标准化

0 均值标准化，也叫作正态归一化或者标准差归一化，一般用于将正态分布的数据转换为标准正态分布。通过表达式

$$X_{\text{norm}} = \frac{X - \mu}{\sigma}$$

处理的数据，满足 $N(0,1)$，也就是平均值为 $\mu = 0$、标准差为 $\sigma = 1$ 的标准正态分布。

不过，这种归一化计算有其局限性。如果输入的对象 X 不符合正态分布，就不适合用这种方法来处理。

3. 非线性归一化

前面介绍的两种归一化方法覆盖了 99% 的工程应用场景，其他归一化方法可以作为补充，在应用中占比非常低。

例如，可以用 log（对数函数）、exp（e 的指数函数）、tan（正切函数）进行处理。这几个函数的特点比较明显，就是定义域范围变化和值域范围变化严重不对等。对数函数 log，x 的变化快，y 的变化慢，exp 和 tan 两者则与 log 相反。根据这种特性，log 适合用在 x 在正数范围内分布范围较大的场景中，exp 和 tan 适合用在较小的范围内彼此之间贴得很“紧”的那些 x 上（通过这两个函数，可以把它们之间的距离拉大一些）。

6.12.4　超参数的设置

超参数（Hyper Parameter）通常是指那些在机器学习算法的训练过程开始之前设定的参数。这些参数的值通常是没有办法通过算法本身学会的。与其相对的，就是在算法中可以学会或学到的那些参数权值 w 和偏置 b。

超参数在深度学习中的使用场景是比较多的。一个网络应该有多少层，这本身就是一个超参数设置问题；一个网络中的各种参数应该以什么样的方式初始化，也是一个超参数设置问题。此外，学习率 η 及每个 Batch 的大小为多少，都是超参数设置所研究的问题。

超参数的设置，一直以来都是深度学习研究中难度非常高的一个领域。就拿网络中的各种参数应该以什么样的方式初始化这个问题来说，到现在为止仍然莫衷一是，每年都有很多讨论这个

问题的论文发表出来。以前我们奉为经典的方法，是以 $\frac{N(0,1)}{\sqrt{d}}$ 的方式实现参数初始化，$N(0,1)$ 就是让整个网络中的 w 和 b 以 0 为平均值、以 1 为标准差来做正态分布，d 是每一层的神经元个数。一种假设是：因为各种物质在统计学中都是符合正态分布的，所以，用这种方式进行初始化，应该与“标准解”最为接近，收敛速度更快，网络工作效率更高。尤其在以 Sigmoid 为激励函数的网络中，会让输入在 0 附近，而这不容易让网络达到饱和，导致无法继续训练。

后来，涌现出了 MSRA、Xavier 等初始化方法。在一开始，我觉得参数初始化问题是一个有定论的问题，随着时间的推移，大家的认知会逐渐一致。可是，在读了更多的论文之后，我发现大家的观点分化还是挺严重的，尤其随着网络种类的变化，会有更多的初始化方法可以对某个或者某类网络的初始化进行优化和调整——变成了 6.5 节打比方时提到的老中医养成过程……这让我有点无奈。就像我们在养生类节目中，一会儿听到专家说喝牛奶比喝豆浆好，一会儿听到专家说喝豆浆比喝牛奶好；一会儿听到专家说生命在于运动，一会儿听到专家说生命在于静养……事实上，认知是螺旋上升的，随时都在进步，随时都会被颠覆，谁也无法说清楚哪个才是所谓“真正的真理”。从古至今，莫不如是。

只能说，我自己还是习惯用 $\frac{N(0,1)}{\sqrt{d}}$ 的方式进行初始化。但是，这种方式无法对最终的训练结果产生非常明显的影响——无论是正面影响还是负面影响，都不明显。所以，我的建议是：在研究具体领域或者某篇著名论文所提出的模型时，可以按照相关的观点或者论文的内容来细致地研究——做某个领域的“老中医”也挺好。对广泛的、广谱的问题，不仅要花很长时间去研究，而且很可能得不到结果。要选择什么样的研究方向，就看你的了。

在这么多的超参数里面（也就是在对学习率 η 的认知上），大家的看法还是基本一致的。深度学习中的学习率 η，其实就是前面提到的那个每次挪动的步长基数。η 通常应该取一个比较小的值。这该怎么理解呢？

想象一下（如图 6-91 所示）：当你闭着眼睛，从山谷的顶部向山谷的底部走，寻找山谷的最低点时，是没办法预判最低点的具体位置的。所以，你只能通过向前伸脚试探来挪动自己的重心。如果你感觉身体重心降低了，就说明方向对了。但是，当你走到谷底附近时，就可能出现因为步子太大而迈过谷底的情况。而由于偏导数方向的改变，再次挪动时，就会向谷底的方向挪（只不过因为步子还是很大，所以仍然会迈过谷底）。如此这般，多次折返，还是到不了谷底。

如图 6-92 所示，箭头非常“纠结”，所以此时将 η 设置为较小的值比较合适。

我在做项目时，为 η 取过 0.05、0.01、0.005 等较小的值。至于在你的项目中 η 取什么值更为合适，则需要经过比较来验证。

如果你是从 GitHub 上下载模型来运行的，可以参考模型作者的设定。如果心里没底，还是将 η 的值设置得小一点比较好（虽然收敛速度比较慢，但容易到达更低的位置）。

图 6-91　学习率

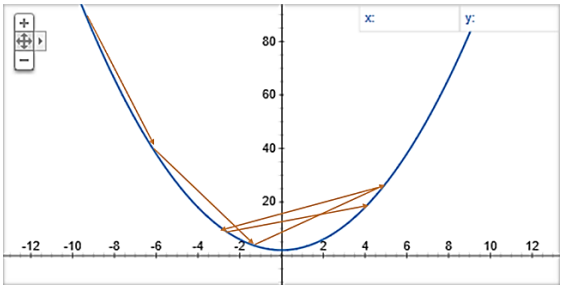


图 6-92　梯度下降

还有一种比较常用的手段，就是分阶段设定 η 的大小。在程序中，通常用变量 `learning_rate` 来充当 η。例如，在程序的开始设定 `learning_rate` 为 0.05，然后进行如下操作。

- 在 Step 1 到 Step 1000 之间设定 `learning_rate` 为 0.05；
- 在 Step 1001 到 Step 2000 之间设定 `learning_rate` 为 0.01；
- 在 Step 2001 到 Step 3000 之间设定 `learning_rate` 为 0.005；
- 从 Step 3001 开始设定 `learning_rate` 为 0.001。

这种阶梯式设定，比一直将 `learning_rate` 设定为 0.05 收敛得稳，比一直将 `learning_rate` 设定为 0.001 收敛得快。具体在哪一个 Step 设定多大的 `learning_rate`，请参考自己的实验结果或者 GitHub 上的优秀工程代码。

6.12.5　正则化

在机器学习领域，很多模型的训练都是非常困难的。有一个明显的现象：参数越多的模型，驾驭起来越难。简单地说，就是“难训”——训练过程不会向着你想象的方向进化。这种现象体现在多个方面，但无外乎就是欠拟合和过拟合。

正则化（Regulization）这个话题，按说不应该由欠拟合这个话题引出，因为它主要针对的是过拟合问题。不过，既然说到这里了，就一起解释一下吧。

1. 欠拟合

欠拟合（Underfitting）是机器学习领域一个现象，是指在一个机器学习模型的训练过程中，最终训练出来的模型在精度上远远没有达到要求的情况。例如，一个用来区分一封邮件是垃圾邮件还是正常邮件的文本分类器模型，对垃圾邮件的识别率仅为 30%。再如，用于识别 MNIST 数据集中的手写数字样本的模型，如果 Accuracy 的值仅为 70%，那显然处于欠拟合状态。

产生欠拟合的原因，通常是模型的参数设置得过少或者不合理。举个例子，你拿到了很多个样本点，将这些样本点标出来，会呈现二次曲线的样子，如图 6-93 所示。

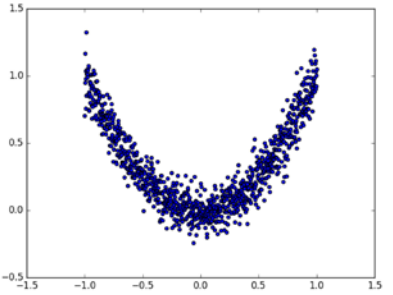


图 6-93 有噪声的 $y = x^2$ 样本

但是，你假设模型为 $y = wx + b$，希望用一条直线完美地穿过这个月牙形的点群，那你就是在难为计算机，因为 w 和 b 无论取什么值，得到的都是一条曲线，而这势必会造成巨大的、无法缩小的误差。这就是典型的欠拟合现象。如果你想将这个模型拟合好，就要设计一个模型 $y = ax^2 + bx + c$，因为用其中的 3 个待定系数进行拟合，才能保证误差足够小。

如果一个处在训练过程中的 MNIST 分类器模型，在某个 Step，Accuracy 的值仅为 65%，而你突然中止了训练，那么此时的模型也是一个欠拟合的模型——远没有达到你对精度的要求。

防止欠拟合的方法是比较简单的。对第一种情况，如果参数的数量不够，在深度学习中你大可不必担心，因为网络中的参数多得是。对第二种情况，你只要把学习率设置得足够小，在训练的时候有足够的耐心，等梯度更新到误差足够小的时候，问题就解决了（这时反倒更容易发生过拟合）。

2. 过拟合

深度学习通过大量的线性或非线性分类器、可导或不可导的激励函数及池化层（在卷积神经网络中会使用这种设计）等，对观测对象的特征进行了自动化的提取。

还是以 MNIST 数据集中的图片为例，由于这些图片把像素级别的向量直接放入神经网络，我们已经不需要对这类数据进行归纳和特征提取了。神经网络有丰富的感知功能，能够把这些最小级别的数值提取出来（例如一个像素的 R、G、B 颜色值），通过训练，逐步发现其中的特征和

规律。神经网络丰富的 VC 维是有这个处理能力的，即能够发现那些在训练数据集中体现出来的共性对网络的刺激，忽略那些在训练样本之间对分类影响微乎其微的维度差异。

这种不需要人类的帮助，由机器提取特征的特性，确实非常吸引人。不过，要想让机器为我们完成这项工作，就要在网络中提供丰富的 VC 维（常用的手段就是增加网络的深度和神经元的数量）。

我们粗略想一个数量级吧。如果一个神经元能够成功区分 2 个分类，分别输出 1 和 0，那么两个神经元理论上最多能够成功区分 4 个（2^2）分类，1000 个神经元就是 2^{1000} 个二分类的分类器——大概是 1×10^{301} 这么多！

但是，如此强大的分类能力，会带来如下两个问题。

一是在复杂的网络中，如此多的 w 早就没有了统计学中的权值和权重的意义，无法得到清晰的物理解释，也就无法有效地进行逆向研究。所以，深度学习的模型训练得再好，也只能当成一个“黑匣子”来使用。例如，把 1000 万个像素级的广告图片 x 和由每幅图片的点击量充当的标签 y 作为训练样本，放到神经网络中进行训练。训练结束后，在网络中得到了很多 w 和 b，而这些 w 和 b 的大小已经无法解释了（起码不能解释为对某个像素的感知权重，因为它们之间的关系太复杂了）。这种无法解释的模型大行其道，有很多人愿意去研究它，主要原因是它在实验中表现良好且展示了较高的泛化性。也就是说，换一个数据集或者场景，仍然能用神经网络的建模思路得到解决方案。

二是这种拥有极高 VC 维的网络能够学到很多东西，包括那些样本中所包含的噪声或者特例信息（这是极为糟糕的）。如图 6-94 所示，这种学习能力通常会导致：把个体性的训练样本放到神经网络中，网络会把每个个体的特点牢牢记下，在训练集上 Loss 函数的值会很小，而识别率很高，但是，换一个样本让网络来识别，就会产生严重的误判。

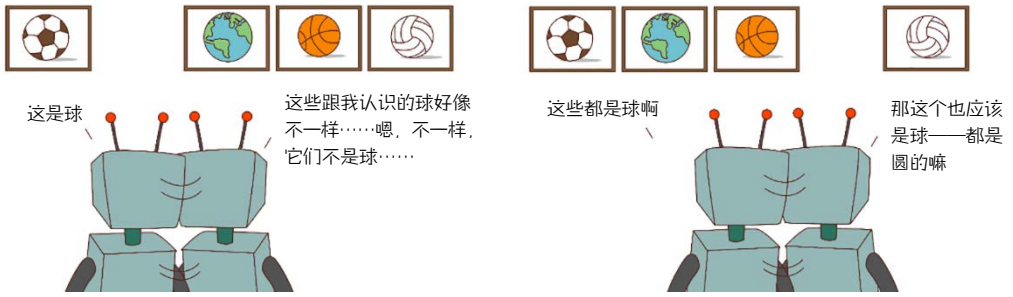


图 6-94　对球体认知的过拟合

这就好比我们教孩子看图说话：给他看的东西越多，尤其是把同样的东西多给他看几种，他自己就会进行归纳和总结，忽略样本之间的差异，找出共性，提高泛化性；如果只给他看数量极少的样本，误判的可能性就会增加，泛化性就会降低。也就是说：如果你只给孩子看一张小轿车

的照片，告诉他这叫汽车，他也确实记住了，当他出门后，看到一辆水泥罐大卡车，他就会觉得疑惑，因为这个东西和你教给他的“汽车”的样子相差实在太大了；如果让孩子多看一些汽车的图片（小轿车、皮卡、SUV、面包车、卡车、吊车、洒水车等），他就会记住它们的共性——有若干个轮子、有玻璃窗、有车灯……这样，孩子再见到其他车辆，可能就会顺利识别了。

看，过拟合的原因和预防方法，我们都已经总结出来了。原因是：样本过少，不足以总结和归纳其共性；参数过多，会因为记住了太多的特性（细节）而误把它们当成共性。在深度学习中，解决过拟合的方案还是有不少的，增加样本数量这种手段就不提了，接下来介绍的手段，都是在模型和训练过程中进行预防或者避免的方法。

3. 训练集与测试集

在以深度学习为技术基础的工程实现中，通常会把所有的样本数据放到如下三个数据集中。在这里，我们正式介绍一下数据集。

- 训练集（Training Set）：用来学习的样本集。使用这些样本，通过网络进行正向传播，得到损失函数，进而通过梯度下降的方法确定网络中的各个待定系数。
- 验证集（Validation Set）：用来验证当前的网络模型中是否存在过拟合问题的样本集。在训练过程中，网络模型通常会在每个 Step 后立刻在验证集中进行验证，我们会同步观察到，在验证集上模型的表现如何、损失函数的值是否会减小、准确率是否有所提高。
- 测试集（Test Set）：在训练结束后，为测试模型的能力（不论是分类模型，还是回归模型）而设置的数据集。

训练集的样本数量在所有样本中占大头。在我接触的项目中，训练集的样本数量通常占样本总数的 80% 左右。验证集和测试集都相对比较小，样本数量大约为样本总数的 20%、10% 甚至更少。也就是说，如果我们拿到 10000 个样本，要进行一次模型训练，可以用 8000 个样本做模型训练，用 1000 个样本做同步验证，用 1000 个样本做测试。更常见的操作方式是：拿到 10000 个样本，用其中的 8500 个做训练，1500 个做验证，不设测试集（在训练过程中，只要由 8500 个样本组成的训练集的正确率和由 1500 个样本组成的验证集的正确率基本相等，且精度能够满足业务需求，训练就可以结束了）。

验证集是在深度学习中为数不多的用于预防过拟合的手段之一。也可以说，验证集是深度学习训练过程的标配。在 PyTorch 中，允许在训练的同时在验证集上使用模型，即一个进程在训练样本的过程中不断更新模型，另一个进程把验证集中的数据实时输入模型并进行测试。训练集的 Loss 函数的值是不断降低的，而准确率 Accuracy 的值是不断升高的（因为在训练中会为了降低 Loss 函数的值而不断进行调整，学到更多、更深层的信息）。

但是，在测试集中，你几乎一定会看到：Loss 函数的值在减小到一定程度后开始增大，或者准确率 Accuracy 的值在上升到一定程度之后开始下降。这个“拐点”就是过拟合的开始，要在这一点终止训练。简言之，因为在深度学习模型中非常容易出现过拟合问题，所以，如果拿到了

大量的样本数据，别把它们都用来训练，一定要拿出一部分来做验证。

为了预防过拟合，研究深度学习的数据科学家们通过实验和研究找到了一些办法，我们来看看都有什么。

4. DropOut

DropOut 是深度学习训练中较为常用的方法，主要目的是克服过拟合问题。由于整个网络中的系数非常多，网络的记忆能力非常强，很多没有泛化能力的个体性特征也会被网络记住，这就容易使那些在训练阶段被网络记住的特性无法通过验证集的验证，我们就会看到损失函数的值在训练集中下降、在验证集中上升的“奇怪”现象。

DropOut，顾名思义，就是“丢弃”。丢弃什么呢？在一轮训练中，丢弃一部分网络节点。例如，可以在某些层上临时关闭一些节点，让它们既不接受输入，也不进行输出——相当于整个网络的结构发生了改变。在下一轮训练中，再有选择地临时“关闭”一些节点（原则上是随机的）。这个过程，如图 6-95 所示。

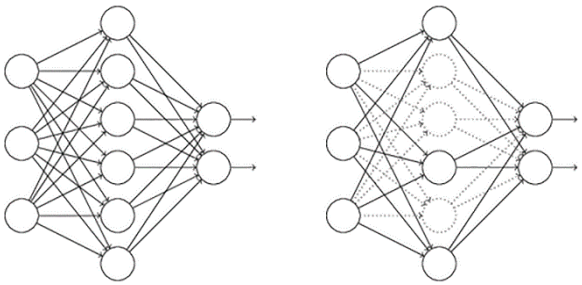


图 6-95　DropOut 示意图

这样一来，相当于在每次训练中将网络的一部分作为一个子网络或者子模型。这种方法在一定程度上减少了参数的数量，降低了产生过拟合的风险。在最终的分类阶段，将所有节点置于有效状态，就可以把训练中得到的所有子网络“并联”使用，形成一个由多个 VC 维较低的分类模型组成的完整的分类模型了，如图 6-96 所示。

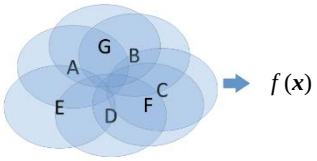


图 6-96　DropOut 原理

DropOut 的应用非常广泛，且主要用在全连接层中。在我见过的项目中，还没有 DropOut 用在全连接层以外的情况。

用 PyTorch 设置训练中的 DropOut 比例是一件非常容易的事情，我们参考如下代码片段。

```
net = torch.nn.Sequential(
    torch.nn.Linear(10, 200),
    torch.nn.Dropout(0.5),
    torch.nn.Sigmoid(),
    torch.nn.Linear(200, 100),
    torch.nn.Dropout(0.5),
    torch.nn.Sigmoid(),
    torch.nn.Linear(100, 1),
)
```

这是一段典型的在 PyTorch 中定义“顺序类型”的代码。“顺序类型”是指网络中没有奇奇怪怪的分叉或交叉，数据从前到后鱼贯而出的标准的“串联”形式。这种“长”得比较标准的网络，都可以使用 `torch.nn.Sequential` 这个类容器来搭建。

在这个例子中，我们一共定义了 3 层网络，具体如下。

- 第一层是一个 10×200 的全连接层，每个神经元在训练中有 50% 的概率被丢弃，激励函数是 Sigmoid。
- 第二层是一个 200×100 的全连接层，每个神经元在训练中有 50% 的概率被丢弃，激励函数是 Sigmoid。
- 第三层是一个 100×1 的全连接层。

网络的定义还是非常简单的。这种形式很像在 TensorFlow 中使用 Keras 的方法。这样看来，PyTorch 也算博采众长了，虽然起步晚，但可以充分吸收各种框架和工具的优点。

5. 参数范数惩罚项

参数范数惩罚项是一种非常重要的正则化手段，不仅在深度学习中，在传统的机器学习中也有应用。从字面上确实不太容易理解参数范数惩罚项的正则化到底要干什么，其实，它的意义非常简单。

在机器学习中，我们是通过将大量样本放入模型中进行训练的方式得到待定系数的。然而，不论哪种模型，我们都希望它在精确的前提下尽可能简洁。请注意，这里的“精确”可不是说在测试集上精确就够了，而是指其泛化能力要强，也就是说，在验证集和其他测试集上都要有好的表现。

对于归纳和认知，我们都有这样的经验：观察到的各种认知对象，越是能够描述其共性的东西，就越抽象、越简洁，泛化性越高；反之，越是用于精确描述特性的东西，“个性化”的特点通常越明显、越具体、越复杂，泛化性越低。无论是归纳和描述人、猫、狗等动物，还是飞机、轮船、汽车等机械设备，甚至球体、物质等更为抽象的对象，泛化性越高的东西，就越抽象、越简洁。而正则化这一过程，就是帮助我们找到更为简洁的描述方式的量化过程。

$$C = C_0 + \frac{\lambda}{n}\sum_w |w|$$

就是改造后带有参数范数惩罚项的损失函数。在前面我们已经了解了损失函数，不过我们提到的损失函数其实只有 C_0 部分，而不包含后面的部分，在卷积神经网络的实验中，才提到了正则化项。

从学术的角度看，损失函数的前半部分可以称作“经验风险”，后半部分（也就是增加的参数范数惩罚项部分）可以称作“结构风险”。

所谓“经验风险”，是指那些由拟合结果和样本标签之间的残差总和产生的经验性差距带来的风险。差距越大，模型拟合失效的可能性就越高，这当然是风险——欠拟合的风险。所谓“结构风险”，就是我们刚刚提到的那个概念了。因为我们希望这种描述能够尽可能简洁，以保证较高的泛化性，所以要添加

$$\frac{\lambda}{n}\sum_w |w|.$$

这个因子的含义是，把模型中所有的权重 w 的绝对值加起来，除以样本数。其中的 λ 是一个权重，也可以称为正则化系数或惩罚系数，用于表示网络对这部分有多“重视”。如果比较重视结构风险，或者不希望结构风险太高，那就设置一个较大的 λ，使整个损失函数向着权值 w 减小的方向快速移动。换句话说，w 的值越大，整个因子的值就越大，网络就越不“简洁”。

这种正则化因子叫作“L1 参数范数惩罚项”（下称“L1 惩罚项”）。常用的还有一种叫作“带有 L2 参数范数惩罚项”（下称“L2 惩罚项”）的

$$C = C_0 + \frac{\lambda}{2n}\sum_w w^2.$$

二者在形式上非常接近，只不过带有 L2 惩罚项的 w 是在平方后才做求和运算的。当然，在这种情况下就不存在取绝对值的问题了，因为平方运算的结果肯定是非负数。

带有正则项的损失函数的导数，当然要和只有经验风险项的损失函数有所区别。带有 L1 惩罚项的导数为：

$$\begin{aligned} C &= C_0 + \frac{\lambda}{n}\sum_w |w|; \\ \frac{\partial C}{\partial w} &= \frac{\partial C_0}{\partial w} + \frac{\lambda}{n}\mathrm{sgn}(w); \\ (w)^n &= \left((w)^{n-1} - \eta\frac{\partial C_0}{\partial w} - \eta\frac{\lambda}{n}\mathrm{sgn}(w)\right). \end{aligned}$$

除了经验风险项对 w 求导所贡献的部分，还有结构风险项对 w 求导所贡献的部分。$\mathrm{sgn}(w)$ 表示取 w 的符号，大于 0 就是 1，小于 0 就是 –1。

带有 L2 惩罚项的导数为：

$$C = C_0 + \frac{\lambda}{2n}\sum_w w^2;$$

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n}w;$$

$$\frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b};$$

$$(w)^n = (w)^{n-1} - \eta\frac{\partial C_0}{\partial w} - \eta\frac{\lambda}{n}w.$$

L2 惩罚项中的 $\frac{\lambda}{2n}\sum_w w^2$，在求导后正好可以消掉 $\frac{\lambda}{2n}$ 的分母中的 2，计算起来要方便一些，这也是在构造这种因子的时候特别设计的 Trick。

Trick 一般翻译成“窍门”“技巧”“花招”，总之让人感觉有点“不地道”。但是，在 PyTorch、TensorFlow、MXNet 等框架的实现中，众多世界顶级研究人员经过潜心研究，在很多处理细节上，通过增加不同的 Trick，对局部的实现进行了优化，从而改进了精度、计算速度、实现复杂度等方面的问题。这些 Trick 的基本思路大都来自学术界的基础经典教材和论文。然而，完全按照这些资料或者数学推导的方式去做，实现起来会有诸多不便，所以这些 Trick“取巧”地出现了。

我们从可视化的角度看一下正则化的实现过程，如图 6-97 所示。

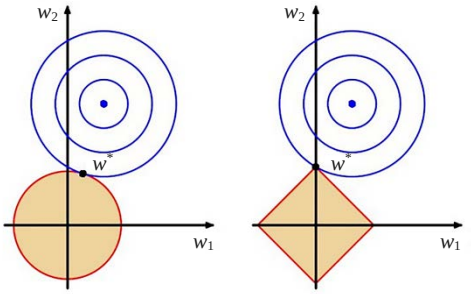


图 6-97　L2 和 L1 惩罚项

假设在一个模型中只有w_1 和 w_2 两个维度作为待定系数，“理想解”在圆心的位置。当然，这里画出来的解在第一象限，但实际上，它也可能出现在其他位置。由于初始化时 w_1 和 w_2 可能会在其他位置，在训练过程中会逐步从这个初始化的位置向圆心靠拢。

圆心周围一圈一圈的线，其实是损失函数等高线。也就是说，当由 w_1 和 w_2 组成的坐标点 (w_1, w_2) 在这一圈上的任意位置都会产生大小相同的损失函数，而初始化位置不确定（可能会出现在一圈上的任意位置）时，显然远离坐标系原点 (0,0) 的点 (w_1, w_2) 会产生更大的结构风险，因为其拥有更大的 w_1 和 w_2 值。下面的圆圈和正方形分别表示由 L2 和 L1 所产生的损失值，左侧

是 L2 的，右侧是 L1 的。

加入正则化项之后，损失将由两部分产生，所以损失函数在收敛的时候要兼顾“小”和“精确”两个特性。可以将经验损失看作“精确”这个特性，它会让解向圆心收敛；可以将结构损失看作“小”这个特性，它会让解向原点收敛。最后，解会出现在两者都比较小的位置，例如图 6-97 中的点 w^* 。如果只有两个 w 的值，是可以画出来的；如果整个空间里有几百万个 w 的值，恐怕就画不出来了。大家可以想象一下，在一个几百万维的空间里 w 被正则化项拉向原点的过程。在一次模型搭建的过程中，通常先不添加正则化项，只用带有经验风险项的损失函数来训练模型。当这种模型训练结束后，再尝试添加正则化项来进行改进。

PyTorch 提供了 L2 惩罚项的功能，但是没有提供 L1 惩罚项的功能。其实，这有点不方便。虽然我们能手动实现 L1 惩罚项的功能，也就是根据损失函数的定义去添加这个惩罚项，在前面配一个小一些的 λ（这个权重值通常都比较小），但根据 PyTorch 的官方文档，如果要使用 L2 惩罚项，应该在优化的时候调用，而不是显式地在损失函数中声明，示例如下。

```
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4, weight_decay=1e-5)
```

其中，`weight_decay=1e-5` 就是调用 L2 惩罚项对损失函数进行惩罚。

说一下 L1 和 L2 两个惩罚项的不同吧。如图 6-97 所示，L2 惩罚项是可以把权值 w 变小一些的，但我们没有把它们尽可能变成 0 的机会，而 L1 惩罚项可以把权值 w 变成 0。L1 惩罚项的切点位置在坐标轴上，这一点 L2 惩罚项可是做不到的。

6. 提前终止

还有一种方法叫作提前终止（Early Stopping），如图 6-98 所示。

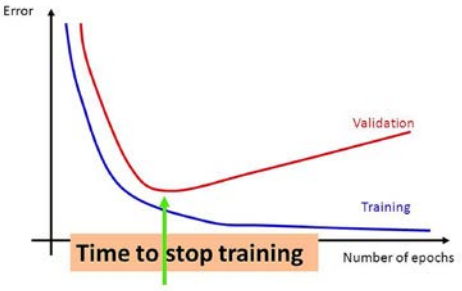


图 6-98　提前终止

当出现过拟合的时候，我们可以看到的最明显的现象，就是在训练集上的准确率持续下降或者降到很低，而在测试集或验证集上的准确率却在升高。如图 6-98 所示，箭头指向的位置就是我们推断出来的发生过拟合的位置。从这里开始，深度神经网络模型会更多地记住训练集中的样本特性，并把它们当成共性。

既然原因如此简单，那么在此刻停止训练就可以了，这就是提前终止的基本思想。

一般我们不会紧盯训练过程，并在终端上观察训练集和验证集上的准确率——这么做有点太不智能了……比较简便的方法有如下两种，供大家参考。

- 方法一：每隔几个 Step（一个 Step 就是一个 Mini Batch 的训练）就进行一次进度保存。这个“进度”，主要是指 Step 的次数序列、模型文件及在训练集和测试集上的准确率。这样，即便我们去做其他事情了，在一段时间以后，回顾训练过程时，仍然可以比较清晰地看到在什么时候发生了过拟合。因此，选择那一刻的模型文件作为训练的模型就可以了。但是，在深度学习的训练中，模型文件通常有几百兆字节（MB），而这可能会占用较多的磁盘空间。
- 方法二：通过类似实时监控的方式，观测在训练过程中验证集上的正确率不再提高后的若干轮，然后停止训练。这若干轮的参数设置，我们习惯上称为“Patience”，也就是“耐心程度”。每个人选取 Patience 的习惯可能不一样，我倾向于选取 5 这种不大不小的数字。如果数字太小，任何样本数据分布的抖动都会导致训练的终止；如果数字太大，就会导致模型敏感度降低，明明已经出现了过拟合，但过了很久才停止训练（也就起不到防止过拟合的作用了）。

在 PyTorch 中，Early Stopping 并没有得到显式的支持，因此我们只能通过编写代码的方式来实现它。如果不想自己编写代码，就只能求助 torchsample 模块了。这个模块之于 PyTorch，有点像 Keras 之于 TensorFlow，不仅 L1 和 L2 惩罚项被支持了，Early Stopping 也被支持了。此外，torchsample 模块支持很多工具函数。我觉得，初学者可以关注一下。

6.12.6 不唯一的模型

在你兴冲冲地使用深度学习模型进行训练时，可能会发现一个问题：即便训练样本都是一样的，第一次训练出来的模型和第二训练出来的模型经过同样的轮数，得到了相近的损失函数 Loss 的值和准确率 Accuracy 的值，模型结构也很可能是不一样的。

原因比较简单——在整个训练过程中，随机因素很多。

在随机梯度下降算法中使用 Mini Batch，相当于从众多数据中选择一部分来进行模型的训练，那么，从 Loss 函数的值较大的位置会向哪个方向挪动，就和每一批被选入的样本特征有关了。虽然我们知道，在统计学中，这些样本的共同特征会帮助网络记忆、学习相关的提取技巧，但从数学的角度看，这种收敛的效率和方向会导致模型结构的不同。

除此之外，在网络初始化时，权值是随机初始化的。在不同的训练过程中，不同神经元的权重会被初始化成不同的值，这同样会影响不同神经元的收敛方向。

换句话说，这些因素会导致用同样的训练数据训练出来的网络模型可能不唯一（如图 6-99 所示）。然而，即便这样，我们仍能发现，在同一个数据集上用同样的网络模型进行训练，能够得到极为近似的准确率 Accuracy 的值。这是因为，整个网络提供的拓扑结构包含的关系很丰富，可以

提供不止一种实现最终模型的方式——所谓“条条大路通罗马”。在训练过程中，网络中的每个神经元每次的收敛方向虽然可能不一致，但仍能在迭代中形成新的逻辑自洽性，并最终完成训练。

图 6-99　不唯一的模型就像池塘里的叶子

6.13　深度神经网络的发展趋势

深度神经网络的发展是从 1986 年的 LeNet 开始的，如图 6-100 所示。

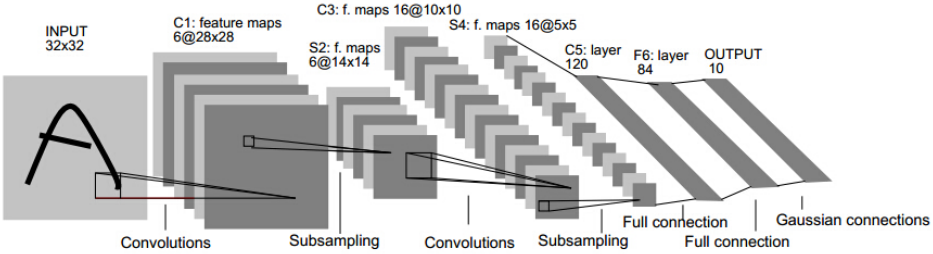


图 6-100　LetNet 结构

要知道，那个时候还是 386 作为主流 PC 的时代，CPU 是单核的，频率只有 33MHz，而当下主流的 Intel i7 8700 CPU 的 6 核 12 线程、4.6GHz 的计算能力，是 386 的 1600 倍还要多！可以说，386 还不如现在的一块智能电子表的计算能力强。在那个年代提出这样的模型，应该说是非常有想象力的。就好像在农业文明伊始，人类还在用牛拉犁耕地的时候，就有人想：如果有一天，有头全自动的“铁牛”，一天能犁 2000 亩地……

2012 年，AlexNet 出现后，人们才第一次知道深度学习这么“深”、效果这么好。AlexNet 是一个典型的卷积神经网络，由 5 个卷积层和 3 个全连接层组成，如图 6-101 所示。在 2012 年，AlexNet 在 ILSVRC（ImageNet Large Scale Visual Recognition Competition）比赛中获得了冠军，其 Top-1 错误率仅为 37.5%（2010 年该比赛冠军的此项指标为 47.1%，这是一个飞跃）。

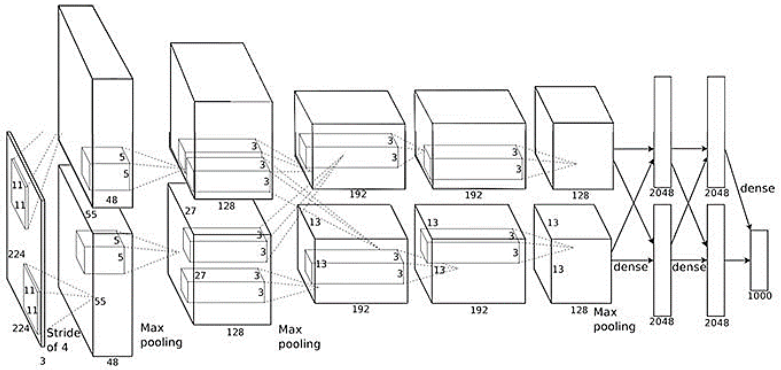


图 6-101　AlexNet

在此之后，因为人们通过新一代的模型得到了很多启示和鼓舞，对深度学习的增强手段就开始变得多样化了，出现了多个流派，如图 6-102 所示。

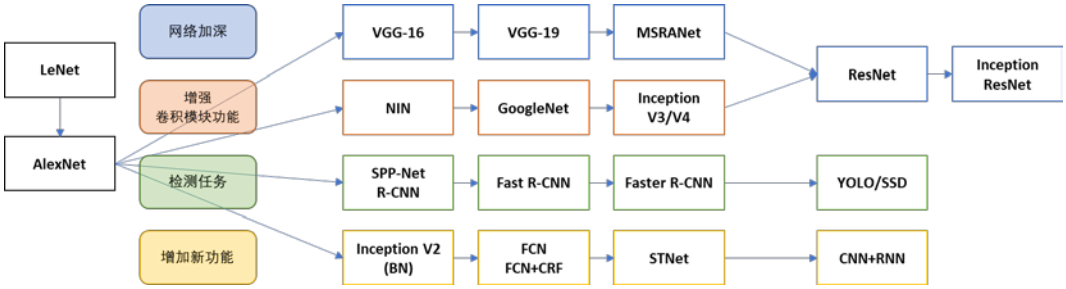


图 6-102　深度学习流派的分化

例如，牛津大学计算机视觉组（Visual Geometry Group）先后设计了 VGG-16 和 VGG-19 等模型。在 VGG-16 中有 16 个 Layer。在 VGG-19 中有 19 个 Layer，既有卷积层，也有全连接层，如图 6-103 所示。从结构上看，很多卷积层叠加在一起，但除此之外，和前面介绍的普通卷积神经网络没有太大的区别。更多的层，意味着更为丰富的内容提取，以及更为犀利的分类能力。

再如，采用 NIN（Network In Network）的设计思想，并按照这一思想发展出来的网络。NIN 思想是指，对深度学习的增强，不再是把网络加深、再加深这么简单，而是尝试把网络编织成一个个小的网络单元，将这些网络单元叠加，组成一个深度网络。

典型的 NIN 就是由 Inception 单元组成的 GoogleNet（如图 6-104 和图 6-105 所示）。先被设计出来的是 Inception 单元。这样的单元不再是由普通的卷积层叠加而成的，而是由不同尺寸的卷积核加上池化层组成的小网络单元（NIN）。把这些层中的 Inception 叠加起来，形成一个更深的网络。这样做的效果非常好——在 2014 年 ILSVRC 比赛的分类任务中，GoogleNet 以 Top-5 错误率 6.67% 一举夺魁（同年 VGG-19 的同一指标为 7.32%），而参数的数量仅为 AlexNet 的 1/12。

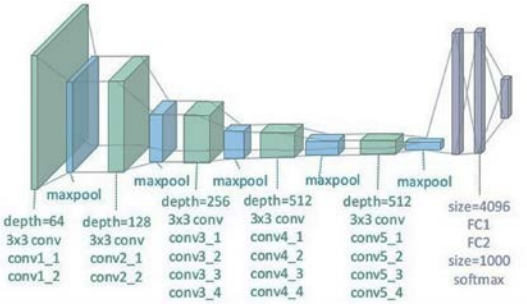


图 6-103　VGG-19 网络结构

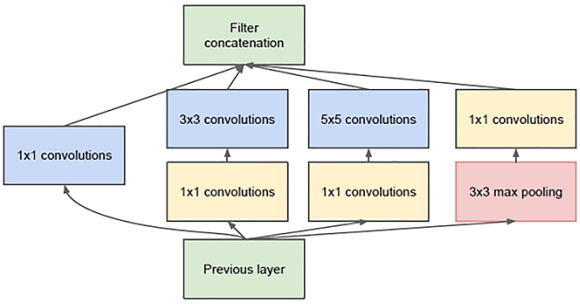


图 6-104　Inception 结构

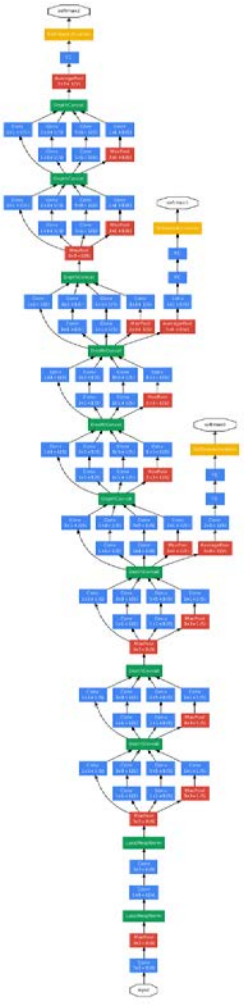


图 6-105　GoogleNet 结构

NIN 网络达到一定深度，就出现了像深度残差网络这样的“巨兽”。2015 年，由微软亚洲研究院的何恺明等人研发的残差网络，能够把神经网络叠加到 152 层之多，并获得了 ImageNet 上的 Classification（分类）、Detection（检测）和 Localization（定位）任务的冠军。

残差网络为什么能有这么大的进步，变得深度如此之大、效果如此之好呢？主要因为残差网络中有一个叫作“Shortcut Mapping”的功能，也就是加入了一个类似“短路线”的东西（如图 6-106 所示）。

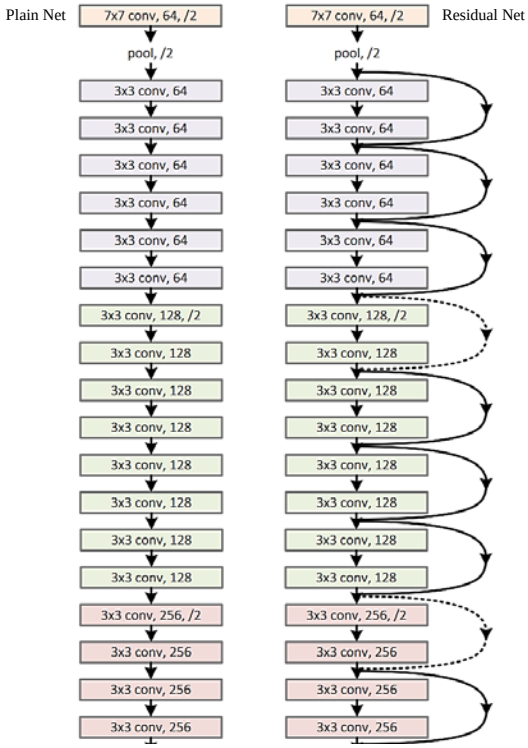


图 6-106　Plain Net 和 Residual Net 的对比

在这种结构的影响下，每个卷积层的输入不仅来自前一层的 Feature Map，也来自两层之间的 Feature Map。

将“短路”层引入后，会出现一个有趣的现象：产生一个非常平滑的正向传递过程。x_{l+1} 和其前一层 x_l 的关系是纯粹的线性叠加关系。如果进一步推导 x_{l+2} 及以后层的输出，会发现展开后如图 6-107 所示，也就是说，后面任何一层的 x_L 的内容，会有一部分由其前面的某一层的 x_l 线性贡献。

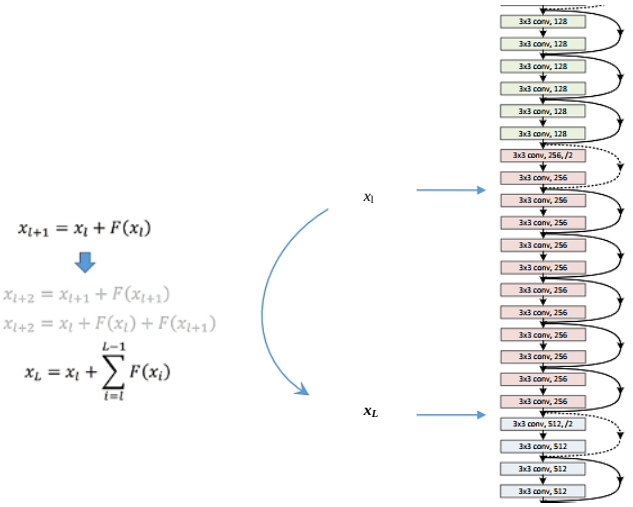


图 6-107　Residual Net 的残差传递

反向的残差传递也是一个非常平滑的过程。在图 6-107 中，某层的输出 x_L 的表达式为

$$x_L = x_l + \sum_{i=l}^{L-1} F(x_i),$$

那么，残差定义为 E（就是 Loss），因此应该有

$$E = \frac{1}{2}(x_L - \boldsymbol{x}_{\text{lable}})^2.$$

$\boldsymbol{x}_{\text{lable}}$ 表示在当前样本和标签已经给定的情况下，x_L 所对应的理想向量值（残差 E 就用来表示它）。接下来，又是老生常谈的求导过程了。在这里可以用链式法则直接求导，很简单：

$$\frac{\partial E}{\partial x_l} = \frac{\partial E}{\partial x_L}\left(1 + \frac{\partial \sum_{i=1}^{L-1} F(x_i)}{\partial x_l}\right).$$

用“白话”解释，就是任意层的输出 x_L 所产生的残差可以传递给其前面的任意层的 x_l，这个传递的过程是非常“快”（或者说“直接”）的，在层数增加的时候也不会出现明显的效率问题。

还有一个值得注意的地方：后面这项 $1 + \frac{\partial \sum_{i=1}^{L-1} F(x_i)}{\partial x_l}$，可以使得 $\frac{\partial E}{\partial x_L}$ 到 $\frac{\partial E}{\partial x_l}$ 是一个线性叠加的过程而非连乘，所以不太可能出现梯度消失的问题。

其他增强神经网络的方法，是对网络功能复合化的需求。一个神经网络，既可以解决分类问题，也可以解决回归问题，那能否两者一起进行呢？例如目标识别类问题，如图 6-108 所示，网络的输入是一幅图片，输出是一个标签矩阵。也就是说，要通过图片中的信息来预测图片上哪里有物体，以及这个物体最有可能是什么、概率为多少。

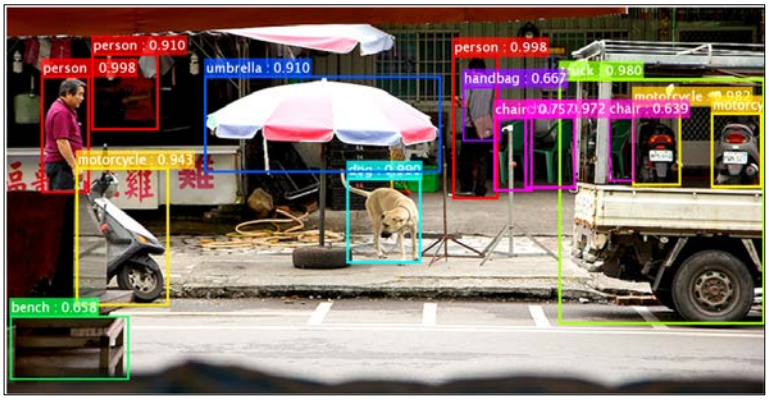


图 6-108 用 Residual Net 进行目标识别

在这样一个题设中，卷积神经网络的训练目的不仅仅是约束映射逻辑、保证分类正确，还包括获取物体位置信息。那么，无论是预测的位置有错误，还是分类有错误，都算是有误差，都应该计入损失函数。

损失函数的形式是

$$\text{Loss} = \alpha \text{Loss}_{\text{reg}} + \beta \text{Loss}_{\text{class}}.$$

其中，Loss_{reg} 是由预测出来的位置偏差而产生的损失值，用 MSE Loss 就可以了；$\text{Loss}_{\text{class}}$ 是由分类产生的误差，用 Cross Entropy Loss 就可以了；α 和 β 分别是对两种损失的侧重程度的配比（误差的量纲不一样，不进行平衡是会出问题的）。

至于 FCN（Fully Convolutional Networks）等网络，就是为了满足一些“非主流”的功能需求而设计的了。FCN 用于像素级别的图片语义识别，例如抠像和背景合成技术。

CNN 和 RNN 的组合应用也比较常见，最典型的就是 ImageCapion 和 Neuraltalk2（输入一幅图片，让网络用一句话把图片的内容描述出来，参见 6.11.2 节）。

后来，深度学习的应用领域更加丰富了，DQN（参见第 8 章）和 DDPG（参见第 9 章）等都是深度学习和强化学习结合应用的例子。

GAN（Generative Adversarial Networks，生成对抗网络）更是以训练方式的变化，让网络不再是用于处理回归和分类问题的复杂函数，而是生成一些东西，如图 6-109 所示。在这种模式下，可以一次训练两个模型，G 模型用于生成，D 模型用于判别。G 模型通常输入一些条件并添加一些噪声，然后将它们映射成一个内容，这个内容可以是图片，可以是文字，可以是任何你想要的形式。D 模型会判断某个输入是否是真实存在的，是否是数据集中的数据。对于 G 模型，我们希望它尽可能强大，希望它生成的信息和数据集中的信息特征是相近的。例如，生成的人脸图片很像真实的人脸图片，生成的花朵图像很像真实花朵的照片。反之，D 模型应该有尽可能强的辨别伪造信息的能力，也就是说，对 G 模型生成的图片，D 模型不应该认为是真实存在的，而应该判定

为假数据。在这个过程中，两个模型的进化方向是相反的，一方的增强会在一定程度上说明另一方的削弱，这样就产生了对抗。对抗的最优状态，应该是达到一种均衡状态——G 模型生成的数据，有一半能够被 D 模型识别，而另一半则成功“骗”过 D 模型。这是双方共同努力的结果。

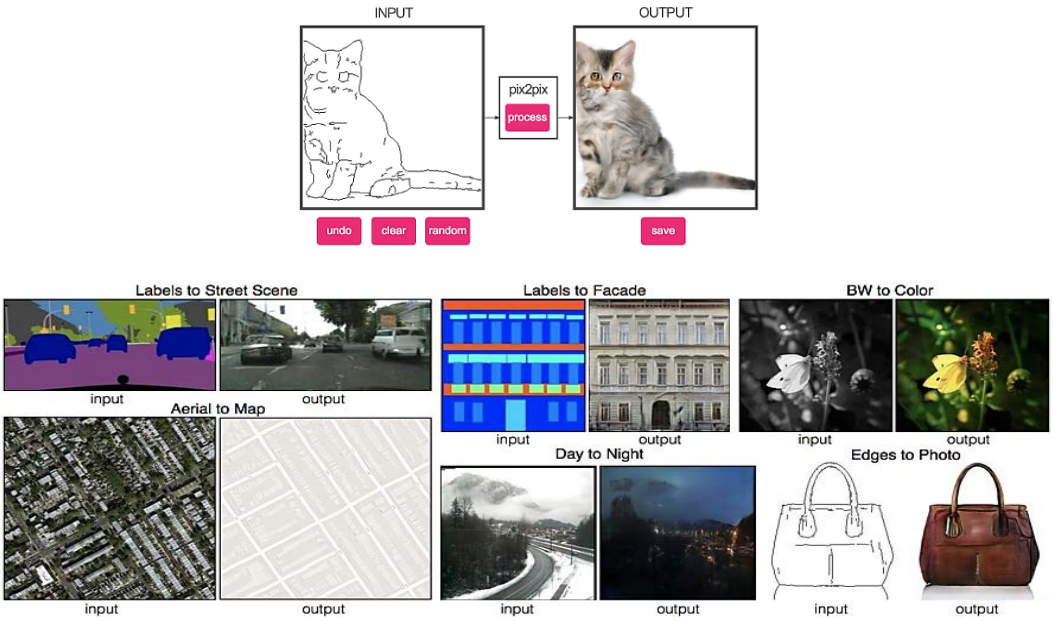


图 6-109　GAN 的效果

训练完成后，通常我们只使用 G 模型来生成想要的东西。例如，输入一幅素描轮廓图片，让 GAN 来着色，从而生成一幅完整的彩色图片（可以用在 CG 或者工业设计中）。

这种功能也可以用来实现超分辨率。例如，输入一幅低分辨率的图片，生成一幅高分辨率的图片。如图 6-110 所示，bicubic 是低分辨率的图片，SRGAN 是用 GAN 生成的高清图片，original 为原始高清图（也就是 Ground Truth）。

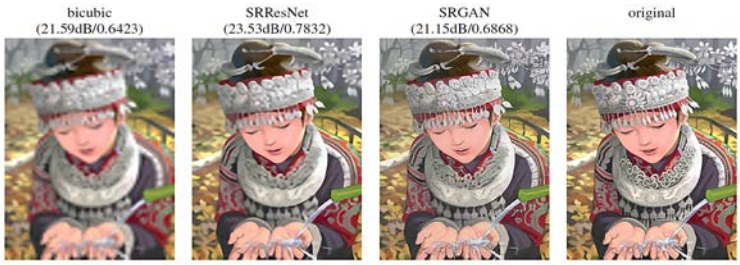


图 6-110　用 GAN 实现超分辨率

因为神经网络本身拥有强大的记忆能力，以及线性和非线性空间的划分能力，所以它能够完成复杂数学问题的学习任务。辅以工程技巧，神经网络可以变得非常深且不易饱和，而这为使用神经网络完成各种复杂的任务打下了基础。我们应该可以预见，在不久的将来，将有更多有趣的深度神经网络应用出现。

6.14　本章小结

你读完了本书中最长且最“浓缩”的一章。

本章覆盖了入门深度学习需要了解的大部分知识，对 PyTorch 的功能和基本使用方法也进行了必要的讲解。

在这里需要强调的是，深度学习虽然有非常好的拟合能力，但说到它的增强能力（或者能够解决的问题），仍旧是本章开头提到的“较强的对分类和回归问题的拟合能力”和“较高的泛化能力”。至于推理、思考和无与伦比的智慧，现在还谈不上。

神经网络的训练过程就是进行正向传播、计算损失、求导数值、用减法更新梯度（如图 6-111 所示），总之，还是普通的加、减、乘、除运算。所以，不要被“人工智能”或者“深度学习”中某些闪光的词汇误导，我们学习的全连接网络、卷积神经网络、循环神经网络，都是以不同的方式对观测到的样本值进行解释或者建模，然后通过实验来验证其在数学层面和应用层面的有效性的过程，并非不可逾越的高峰或者颠扑不破的真理。

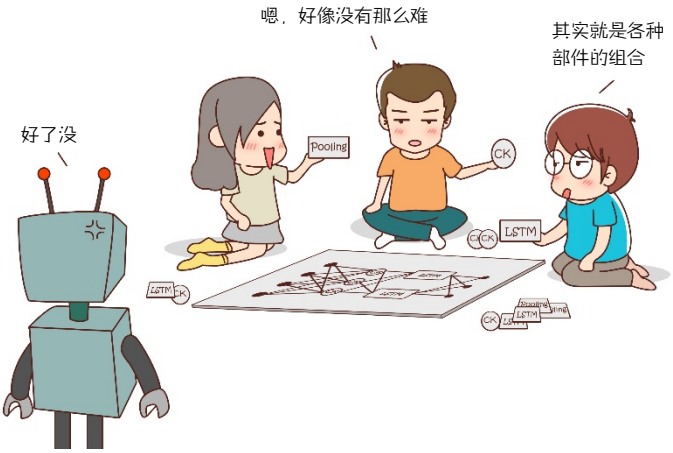


图 6-111　构建智能模型就像搭积木

具体到深度强化学习这个领域，这意味着：在给出状态 s 和 $Q(s,a)$ 的值之后，就可以借助神经网络，比较容易地通过一个模型实现 $s \rightarrow Q(s,a)$ 的映射关系，而不用再去考虑是否要采用诸如查表这样的方法了。从查找效率的角度看，这种方法的处理速度在正向传播的过程中要比查

表快，而且，s 的空间越大，这个特性似乎越明显（因为只需要 CPU 进行一次正向传播就可以实现）。查表则不然，我们不仅需要维护一个表，而且为了在查找的过程中提高效率，需要为这个表设计合适的索引。如果你自己无法解决这些复杂的工程问题，那么你几乎没有选择，只能使用 MySQL、Redis 等第三方数据库。

卷积层的引入，也可以让 $s \to Q(s,a)$ 这个过程变得平滑。卷积层能使网络具有更高的泛化能力，也就是说，将两个相似的场景或者两幅相似的图片输入网络，其中一个场景（一幅图片）的估值被正确估算过，并且在训练中让网络进行了过拟合，那么另一个场景（另一幅图片）即便在训练中没有见过，也很容易让网络对它形成正确的估值。因为它们彼此相似，所以在通过网络后比较容易在最后一层产生极为相近的激励值（也就是得到相似的估值）。这是极好的，泛化性也是由此体现的。

深度神经网络本身能够解决的问题就是这些，它无法帮助强化学习做任何事情。深度神经网络不能提高样本收集的效率，不能帮助 TD 或者 MC 算法实现任何估值 Backup 方面的增强（无论是在效率上，还是在准确率上，都不能），希望大家能够明白这一点。

深度强化学习最终“学习”得如何，在我看来，在很大程度上仍旧取决于强化学习理论范畴中的建模问题，而不取决于深度学习网络。如果非要量化它们的重要性，我也只能拍着脑瓜说一句“三七开”——三分深度学习，七分强化学习。

第 7 章 Gym——不要钱的试验场

在了解基本的强化学习算法和深度学习概念以后，我们已经站在两块坚实的基石上面了。现在我们就可以开始把它们结合起来，做深度强化学习（Deep Reinforcement Learning，DRL）的各种实验了。结合实验，不仅可以理解各种算法模型的强悍之处，还能发现它们各自隐藏的深不可测的“坑”。不过，在此之前，需要了解一个比较重要的概念——给大家介绍一套“玩具”。

在深度强化学习的学习过程中，有一个要素是我们必须要仔细考虑的，那就是环境。环境是供智能机器人活动的小世界。环境要在机器人和它进行互动的过程中，给机器人提供相应的环境状态信息 S 和奖励值 R。即便 R 不能或者不应该由环境给出，状态 S 也必须由环境反馈给机器人。这就非常关键了——如果要熟练掌握深度强化学习算法模型，就需要搭建一个可以完成上述功能的环境。而如果你不擅长搭建环境，或者找不到合适的环境来练习，那你的学习难度就会迅速提高。不过，好在我们有 Gym。

7.1 简介

Gym 是 OpenAI 团队开发的一个主要针对强化学习实验的开源项目。OpenAI 团队，可以说是人工智能界以深度强化学习为代表的独角兽（或者说风向标）级别的团队。OpenAI 团队是 2015 年由马斯克 [①]——对，就是那位以研究如何低成本发射火箭而出名的顶级科技发烧友——与诸多硅谷科技先锋共同商议后创建的。

OpenAI 团队每年都会在各种人工智能顶级峰会上发布多项尖端研究成果，且并不局限于深度强化学习领域。OpenAI 团队的研究员 Alec Radford 就发表过一篇关于深度卷积生成对抗网络 [②]的论文，讨论的是生成人脸合成图像的事情，也就是把人脸的一部分遮住，看能不能用神经网络生成这部分图像。先别管这种技术最终在哪儿能用得上，这件事总归是很有趣的。好多东西在被发明出来的时候，可能人们都不知道应该怎么使用它，或者最初发明它的目的与它最终的应用领域完全不相干。若干年过去，落实到产品上，好像经历了一场“种瓜得豆”的梦。

除此之外，OpenAI 团队有一件事情也是非常出名的，就是在 DOTA2 的 5 对 5 游戏中完全超越人类玩家。这可以说是在 AlphaGo 之后，又一个在智力领域碾压人类的人工智能应用。这个领域的难度，比任何一个传统棋盘类游戏的难度都大得多。要知道，在这类游戏场景中是有“组队”这个概念的，多个智能决策要“协商”出在全局最有价值的战略和战术目标，坚定不移地去执行，

① 埃隆·马斯克（Elon Musk，1971 年 6 月—），企业家。特斯拉公司 CEO，SpaceX 公司 CEO。

② DCGAN（Deep Convolutional Generative Adversarial Networks），深度卷积生成对抗网络。

果断地处理中途发生的任何意外——这得什么样的“脑子”才能完成？我认为，OpenAI 是全球实力最强的人工智能研究团队之一。

另外一个著名的人工智能研究团队就是 DeepMind（深度思考）了。DeepMind 团队位于英国伦敦，由神经认知学博士兼人工智能程序员戴密斯·哈萨比斯[①]创立。2014 年，DeepMind 团队被谷歌收购，成为谷歌旗下一个重要的人工智能研究机构。DeepMind 团队的巅峰之作，就是著名的 AlphaGo。2016 年 3 月，AlphaGo 以 4 比 1 的绝对优势战胜韩国围棋大师李世石九段，在当时引发了对人工智能的大讨论。

两个著名的人工智能团队就介绍到这里。下面我们还是来看 Gym 这个项目。

Gym 的主页为 https://gym.openai.com/，如图 7-1 所示。

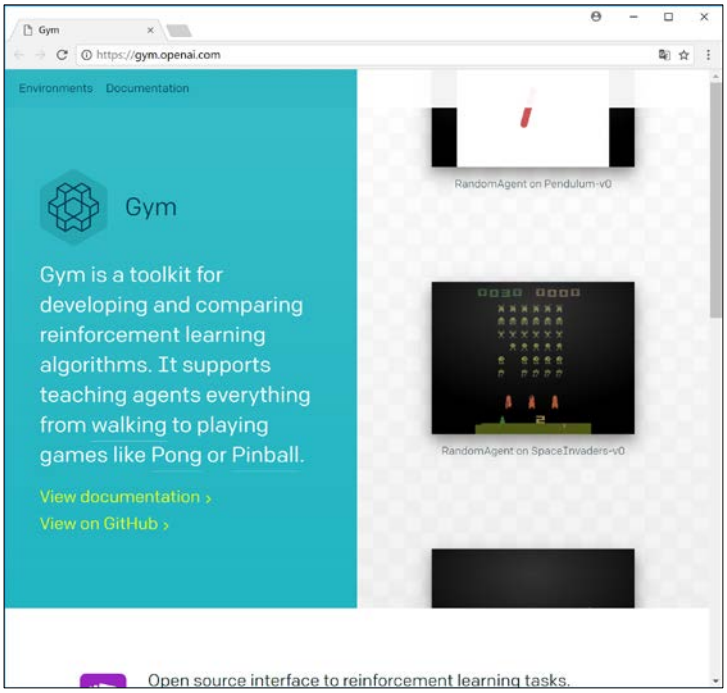


图 7-1　Gym 主页

在这里可以看到对 Gym 涵盖的各种小环境的介绍。这些小环境中的每一个，都是独立的游戏或者看上去像游戏的仿真环境。只需要简单的安装和设置，就可以用程序模拟一个机器人可能在 Gym 环境中做出的各种动作。这个过程通常简单到只需要几条命令和几行代码就能实现。

Gym 项目的开源代码，可以访问 https://github.com/openai/gym 获取，如图 7-2 所示。

① 戴密斯·哈萨比斯（Demis Hassabis，1976 年 7 月—），神经认知学博士，人工智能领域专家，DeepMind 创始人。

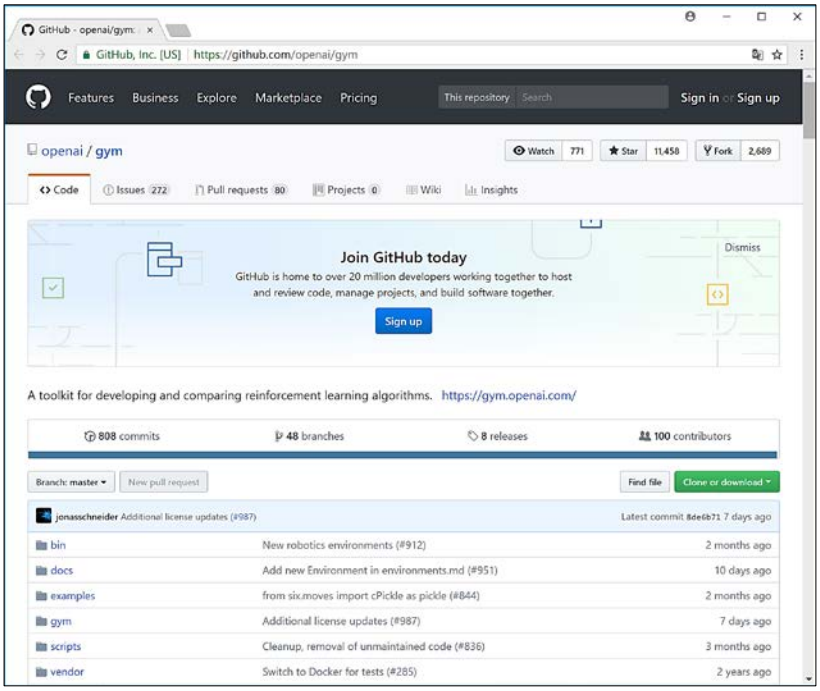


图 7-2　Gym 的 GitHub 主页

在写作本书时候，这个项目已经获得了超过 11000 颗星。对初学者来说，如果只关心深度强化学习机器人应该怎么训练，那就不需要阅读太多的代码，但如果想要研究 Gym 环境的代码，可以把它们下载下来——都是大师写的东西，学一学错不了。

7.2　安装

熟悉开源环境的读者朋友会比较清楚，Gym 的安装方法也不止一种。不过，在本书中，我仍旧采用最简单的 pip 安装方法，具体如下。

```
pip3 install gym
```

在 Python 环境中导入 Gym，做一下测试。

```
python

>>>import gym
>>>
```

如果在执行 `import gym` 命令的时候没有报错，就基本可以认为 Gym 安装成功了。

7.3　类别

打开 Gym 的官方主页，可以看到它封装的一些环境的预览效果。之所以叫作“Gym”，就是希望用户把这个开源工具当成健身房，想怎么玩，就怎么玩。

Gym 的官方网站为了方便用户，专门划分了研究问题的种类（http://gym.openai.com/envs/）。为了称呼方便，我们将它们统称为“小游戏”。

1. 算法问题

第一类是算法（Algorithms）问题，如图 7-3 所示。

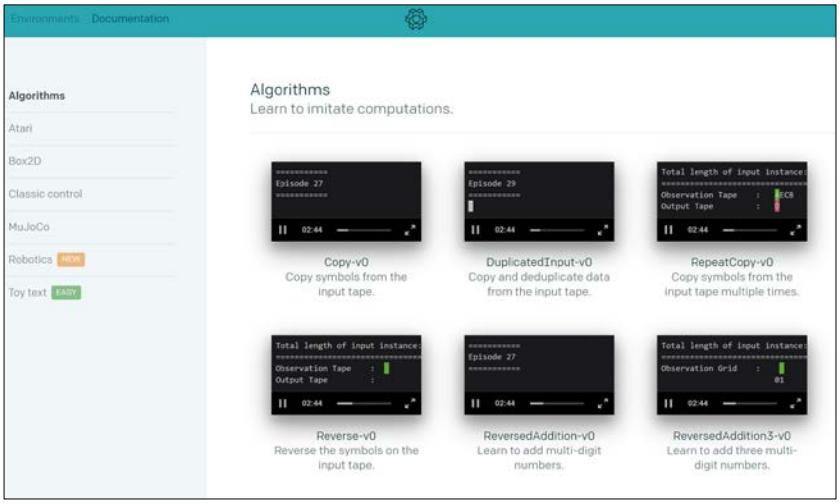


图 7-3　Gym 中的算法问题

以小游戏 Copy-v0 为例，单击图标，可以看到这个游戏的说明：“This task involves copying the symbols from the input tape to the output tape. Although simple, the model still has to learn the correspondence between input and output symbols, as well as executing the move right action on the input tape.”大意是：“这个任务的主要目的是将输入磁盘的内容复制到输出磁盘上。虽然简单，但是模型仍然需要学习输入与输出符号之间的对应关系，同样需要学习在输入磁盘中执行的右移动作。”

这就是让我们以机器学习的方式来学习正确的操作过程的最简单示例了。在这个过程中，机器人只有奖励值可以获取。

2. Atari 游戏问题

第二类是 Atari 游戏问题，如图 7-4 所示。

“Atari”译为“雅达利”，是一款古老的游戏机的名字。Atari 游戏机的历史比“80 后”玩的任天堂红白机还要长，在 20 世纪 70 年代的欧美电玩市场上比较活跃。2013 年，Atari 公司申请了

破产保护，从而永远淡出了历史舞台。然而，Atari 游戏的输入和输出简单，使得训练的难度大大降低。此外，这类游戏的评价方式通常比较简单，以比较游戏得分为主。简单的任务设置，简单的输入和输出，以及其本身的趣味性，用来训练机器人还是非常适合的。很多关于深度强化学习的论文都是基于 Atari 游戏撰写的。

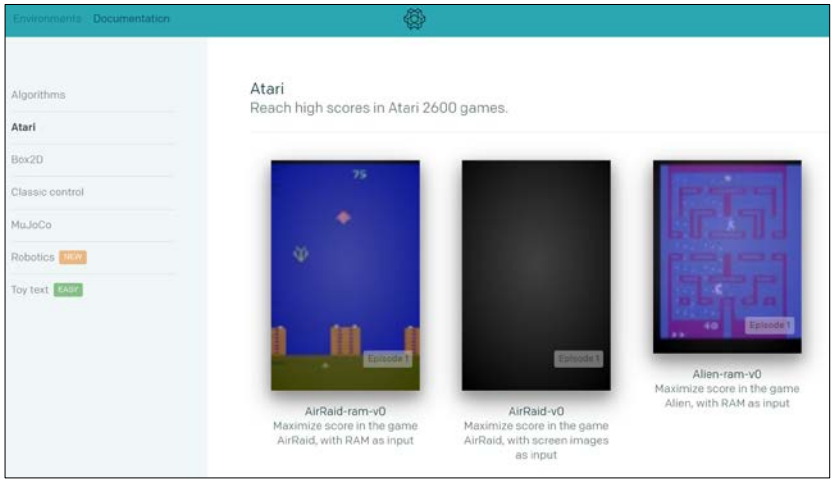


图 7-4　Gym 中的 Atari 游戏问题

3. Box2D 模拟器问题

第三类是 Box2D 模拟器问题，如图 7-5 所示。

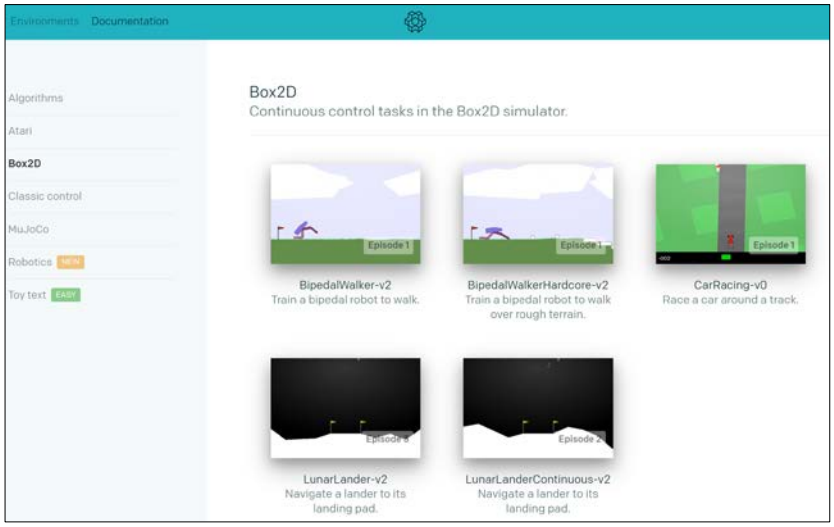


图 7-5　Gym 中的 Box2D 模拟器问题

Box2D 是一类连续控制任务的模拟器。在界面上我们能看到 3 个游戏，一共 5 种不同的题设场景。前两个场景是训练一个双足站立的机器人步行，它们的不同之处是，一个在平坦的地面上行走，另一个在高低不平的地面上行走。第三个场景是一个赛车游戏。另外两个场景是一个月球飞船降落的连续控制任务（引导月球飞船平稳降落）。如果飞行平稳，每个 Step 都能得到正的 Reward 值；如果在途中出现了倾覆、翻滚，或者以很大的速度着陆，就会受到“惩罚”。

4. 经典控制问题

第四类是经典控制（Classic Control）问题，如图 7-6 所示。

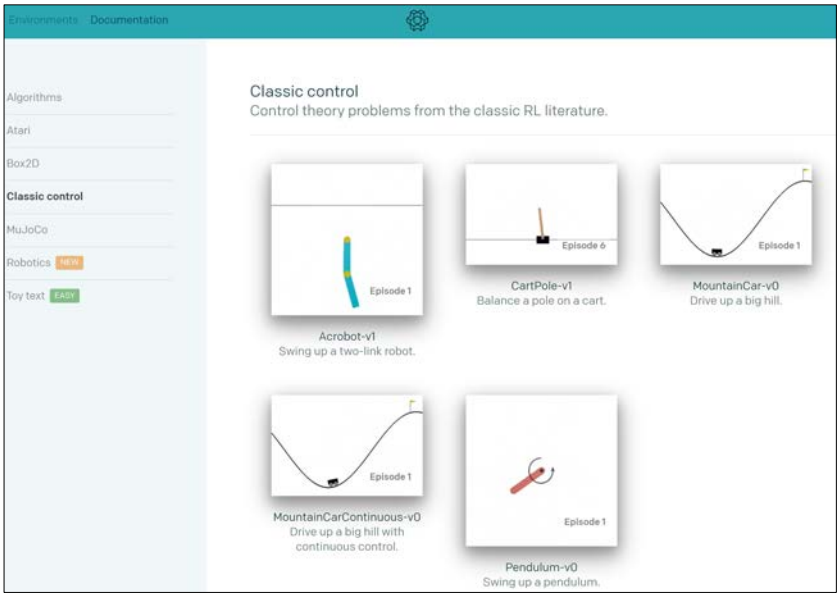


图 7-6　Gym 中的经典控制问题

经典控制问题也是很有趣的，其实就是很久以前各高校、研究所做自动工控时喜欢建模研究的那些经典问题。我们以第二个小游戏 CartPole-v1 为例，它模拟的场景是：在一个一维的水平滑轨上有一辆小车，车上有一根貌似用轴承与其相连的立杆。轴承是可以转动的，立杆在重力作用下会倒下，因此需要让小车左右移动来保持平衡。要想解决这个问题，就需要训练一个机器人，使它能够根据当前小车和立杆的状态来决定是向左移还是向右移，从而调整重心。

5. 多关节接触动力学问题

第五类是多关节接触动力学（MuJoCo）问题，如图 7-7 所示。

“MuJoCo”看起来像一个日语词汇，但其实不是，它是“Multi-Joint Dynamics with Contact”的缩写，译为“关节接触动力学”。乍一看有点“不明觉厉”，不过细看就会知道，在这里模拟的

场景是那种由多个刚体组件组合而成的机器人或类机器人在三维场景中的运动，不仅有头、胸、腹、四肢之间衔接灵活的人，也有两条腿的狗、四条腿的蜘蛛（小游戏名叫“Ant-v2”，但我不觉得那东西像蚂蚁），挑战的项目大都是在重力环境中让这些“智能体生物”正常运动。如果重心出现偏移，它们就会摔倒。而最终目标，是让它们走得远、走得稳——看上去颇为有趣。

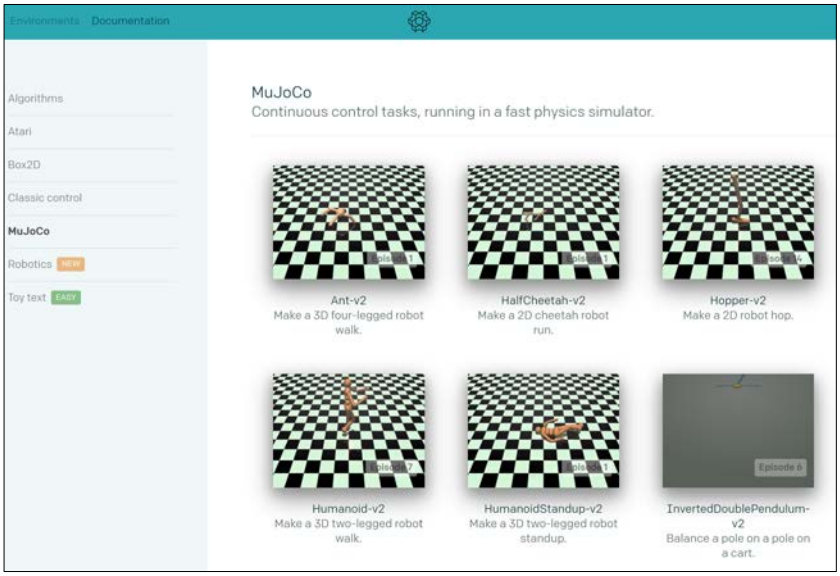


图 7-7　Gym 中的多关节接触动力学问题

我们可以尝试做一些实验，不过 MuJoCo 的环境安装比较麻烦。在本书的附录 A 中专门提供了 MuJoCo 和 mujoco-py 的安装指南。

6. 机器人技术问题

第六类是机器人技术（Robotics）问题，如图 7-8 所示。

这类问题是非常典型的工业控制问题，就是以前我们说的那种在生产线上控制机械臂（或者机器人手臂）的问题。

机器人技术问题中的场景，看上去很有意思。就拿 ShadowHand 这类问题来说吧，如图 7-9 所示，需要训练一只机器人手掌，让它来完成不同的任务。而 HandManipulateBlock-v0 这个小游戏，则需要训练一只机器人手掌，让它把一个方块（Block）移动到指定的位置并以指定的角度放置，这不仅需要准确地放置这个方块，还需要在放置过程中对方块进行适当的翻转。

在机器人技术问题中，其他小游戏的场景与前面介绍的比较类似，一个是移动一个球，一个是移动一根棒，还有一个是简单地让五根手指前往它们各自应该在的位置。

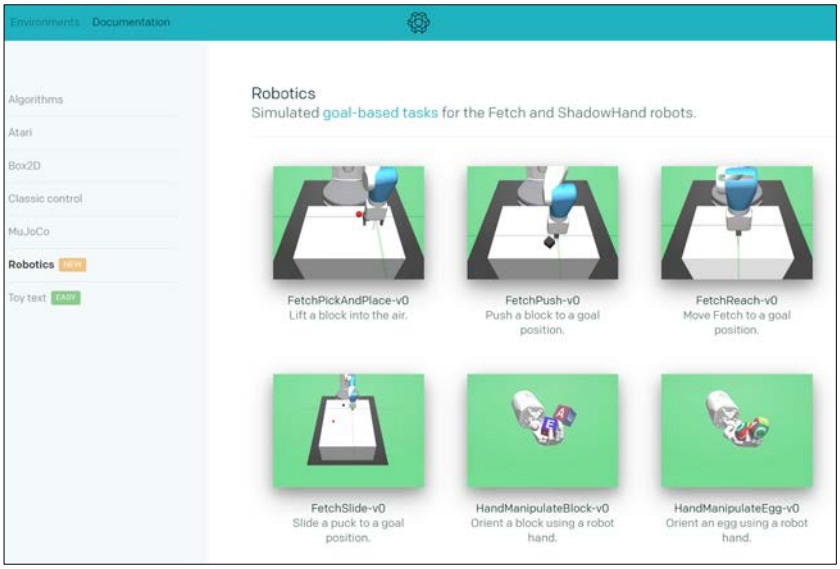


图 7-8　Gym 中的机器人技术问题

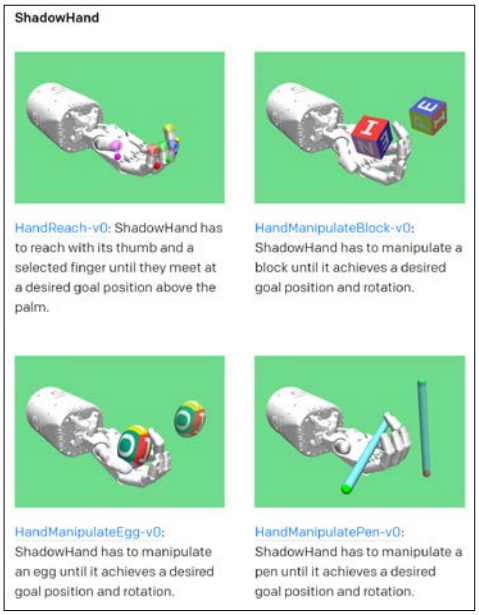


图 7-9　Gym 中的机器人技术 ShadowHand 问题

7. 文本小游戏问题

第七类是文本小游戏（Toy Text）问题，如图 7-10 所示。

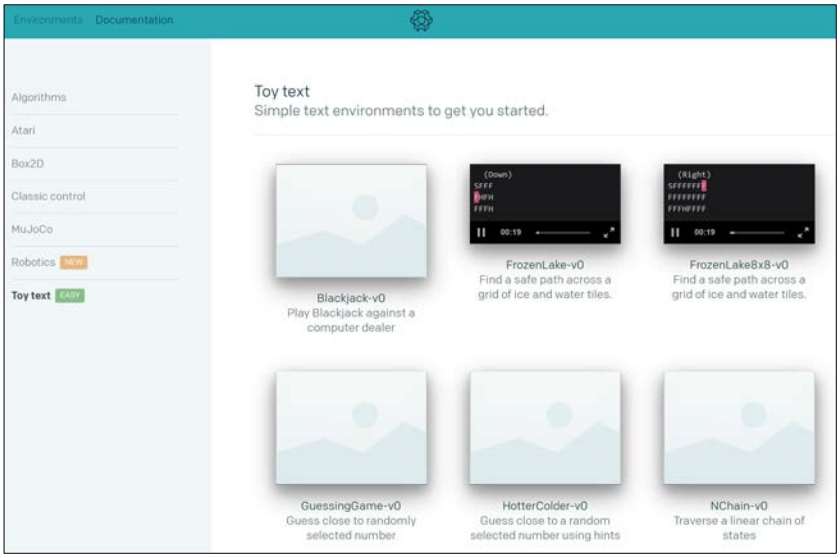


图 7-10　Gym 中的文本小游戏问题

这类问题比较简单——主要是形式简单，输入和输出的维度较低，相对容易训练。不过，这类问题的场景看上去比前面几类问题简单得多，不太容易引起用户的兴趣。如果你的计算机资源不允许你玩别的小游戏，那就拿这类小游戏练手好了。反正，这类小游戏的结构和形式足够简单，即使在训练中出了问题，也可以方便地打印状态并进行排错。这大概是除了游戏代练以外，唯一的玩游戏属于生产任务的工作类型了吧（如图 7-11 所示）。

图 7-11　玩游戏也是搞科研

7.4　接口

Gym 之所以被很多人喜欢，主要是因为接口做得比较好。它对复杂的场景进行了抽象，做了合理的降维，从而把输入机器人的 State 简化到了最简单的程度。同时，它把在一个 State 下做一个动作（Action）后得到一个 Reward 的标量值也定义好了（在大多数情况下是这样的），这简直

太棒了。也许你现在还没有感觉，但等到你做了足够多的实验，在很多以屏幕输入的像素为 State 的实验项目中被“坑”到怀疑人生，甚至替机器人怀疑机器人的人生时，你就会发现，Gym 真是帮了我们大忙，为我们了解人工智能技术做了太多的好事。

Gym 的官方网站提供了一个简易的了解接口的例子（https://gym.openai.com/docs/）。通过这个例子，我们就能知道这个接口的使用方法了。

本书也收录了这段代码，具体如下（你可以在本书示例代码的 ~/DRLPytorch/gym_sample 目录下找到 demo.py）。

```
1 import gym
2 env = gym.make('CartPole-v0')
3 env.reset()
4 for _ in range(1000):
5     env.render()
6     env.step(env.action_space.sample()) # take a random action
```

当你执行这段代码时，可能会很失望，因为机器人就像个醉汉，完全无视那根本该立起来的杆子，驾驶着小车朝某个方向一通跑，直到不见踪影。别着急，我们还没开始训练机器人呢。

第 1 行程序

```
import gym
```

用于导入 Gym 的 Python 接口环境包。

```
env = gym.make('CartPole-v0')
```

这一行通过 `gym.make()`方法构建一个实验环境，`CartPole-v0` 就是小游戏的名字，即构建一个小车立杆子的 CartPole-v0 小游戏环境。如果要训练其他小游戏，只需要把 `CartPole-v0` 改成要训练的小游戏的名字就可以了。每个小游戏都有自己的名字，都可以在 Gym 的官方网站上找到。

```
env.reset()
```

这一行是常用代码，其实就是重置一个 Episode。要想让机器人重新回到游戏中或者开始一盘新游戏，都需要使用这一方法来重置。

对于代码

```
for _ in range(1000):
    env.render()
    env.step(env.action_space.sample())
```

先看循环体里面的两个方法。`env.render()`方法用于做游戏回显，当你使用这个方法后，游戏的场景就会被回显在一个窗口中。在本例中，被我们执行的 CartPole-v0 小游戏中的机器人，就在回显窗口中跑到一边去了，如图 7-12 所示。`env.step()`方法用于提交一个动作，让机器人向环境做一个动作，括号里面的内容就是动作了。

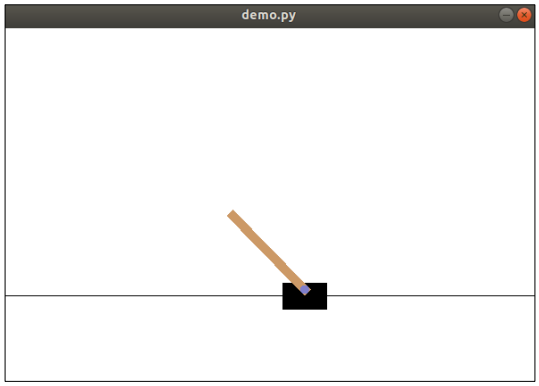


图 7-12 Gym 游戏回显

Gym 中的小游戏，大部分都可以用一个普通的实数或者向量来充当动作。如果觉得不够具象，可以通过本节的例子代码来了解一下输入的是什么。

通过在 env.step()中填充 env.action_space.sample()方法，就能有相对感性的认识了。打印 env.action_space.sample()的返回值，能看到输出为 1 或者 0。env.action_space.sample()的含义是，在该游戏的所有动作空间里随机选择一个作为输出。在这个例子中，意思就是，动作只有两个，一个是 0，另一个是 1，一左一右。

env.step()这个方法的作用不止于此，它还有四个返回值，分别是 observation、reward、done、info。

- observation 是状态信息，是在游戏中观测到的屏幕像素值或者盘面状态描述信息。
- reward 是奖励值，即 Action 提交以后能够获得的奖励值。这个奖励值因游戏的不同而不同，但总体原则是，对完成游戏有帮助的动作会获得比较高的奖励值。
- done 表示游戏是否已经完成。如果完成了，就需要重置游戏并开始一个新的 Episode。
- info 据 Gym 官方描述，是一些比较原始的用于诊断和调试的信息，或许对训练有帮助。不过，OpenAI 团队在评价你提交的机器人时，是不允许使用这些信息的。

在每个训练中都要使用的返回值有 observation、reward、done。但是，observation 的结构会由于游戏的不同而发生变化。以 CartPole-v0 小游戏为例，如果用

```
observation, reward, done, info = env.step(env.action_space.sample())
print(observation)
```

语句进行打印，会看到

```
[ 0.02325978 -0.20024221  0.01490305  0.26784612]
[ 0.01925494 -0.39557364  0.02025998  0.56519207]
[ 0.01134346 -0.59097389  0.03156382  0.86418831]
…
```

这样的数据。这是一个四维的 Observation。在其他游戏中会有维度很多的情况。

别小看 `env.step()`，实际上它完成了一个完整的 $S \to A \to R \to S'$ 过程。我们只要不断观测这样的过程，并让机器在其中用相应的算法完成训练，就能得到一个高质量的强化学习模型。

7.5　本章小结

在这一章中，我们介绍了 OpenAI 团队提供的 Gym 项目。借助 Gym 的内置接口，就可以用最简单的方式实现一个完整的 $S \to A \to R \to S'$ 过程。接下来，我们就可以按照自己的喜好来选择一些小游戏，逐步进行实验了。

Gym 几乎是目前所有开源项目中对强化学习实验环境的封装最为友好的一个了，相信用不了多久，你也会有这样的感触。Gym 也是广大强化学习爱好者最喜欢的一个游戏项目，相当于神经网络界的 MNIST。不过，Gym 比 MNIST 有趣得多。

如果你觉得 Gym 中的小游戏太简单，玩起来不过瘾，希望玩一些更为复杂的游戏，在后面的章节中我们将会介绍，如何借助 SerpentAI 这样的开源项目完成实验，以及一种以未降维的原始屏幕图像或者 App 界面图像作为输入的方法（在理论上，这种方法能够应对的场景确实更多，但是由于维度空间太大，训练的难度也提高了）。作为入门级学习者，以我们目前对算法的了解，Gym 里的这些项目已经足够了。

第 8 章 DQN 算法族

从这一章起，我们就开始正式聊聊将强化学习和深度学习结合起来的各种新玩法。

在本章之前，我们分别讨论了强化学习和深度学习所研究的领域，并了解了它们的长处和短处。毫无疑问，深度强化学习将二者做了结合，既吸收了二者的优点，也集二者的“坑”于一身。也就是说，我们在做任何一个深度强化学习实验的时候，会受到两个领域各自缺点的影响。我认为，这是深度强化学习最大的问题。在这一章中，我们讨论其中理解起来最为容易且论文出现最早的一类模型——DQN 及其变种。我们权且称之为“DQN 算法族”。

DQN 算法族是一个非常有趣的算法族。“DQN”的全称是“Deep Q Network”，其中的“Q”就是指 Q-Learning。从名字上看，这是一种把 Q-Learning 和 DNN（Deep Neural Network）结合起来的模型构架——实际上也是这样的。

8.1 2013 版 DQN

2013 年，DeepMind 的顶尖技术人员在 NIPS[①]上发表了一篇非常著名的论文——*Playing Atari with Deep Reinforcement Learning*。在我看来，这篇论文的内容虽谈不上石破天惊，但足够别开生面，因为这是第一次尝试把 Q-Learning 和 DNN 结合起来，而且貌似很成功。不过，对这篇论文中提出的 Deep Q-Learning Network（DQN）模型，人们褒贬不一。

有的人说，DQN 简直是一个“救世主”般的新概念——如果神经网络能学会自己玩游戏，以后还有什么不能做的吗？当然，这通常是乐观主义者喜欢说的话。

也有人说，DQN 只是将 Q-Learning 迁移到 DNN 上的一次尝试，与将线性回归、逻辑回归、多分类的函数模型迁移到 DNN 上没有本质差别，因此毫无新意。

虽然后者的评价更为苛刻，但我觉得说得在理。下面我们就看看，在这样一个实现过程中，究竟有哪些新东西需要我们学习。

8.1.1 模型结构

每当提到深度学习模型，不论是结构简单的 CNN、RNN、FC，还是由它们组合而成的那些高

① NIPS（Neural Information Processing Systems，神经信息处理系统大会），一个著名的技术峰会。

级的、针对每个具体应用场景的 Deep Speech[①]、Tacotron[②]、Info-GAN[③]等，我都喜欢称它们为“模型”，而不是“算法”。在我看来，“算法”这个词所指代的内容关注的是计算逻辑，是人类用自己的思维方式书写的完整的计算过程。而刚刚列举的这些概念则不同，它们只是映射的逻辑或者结构，是人类摆出来的“架子”，“架子”上放满了待定系数。要想实现它们，需要进行大量的样本训练，并在训练中确定那些待定系数（也就是权重），从而确定计算逻辑。可以看出，两者的区别还是蛮大的。

2013 版 DQN 的结构是非常经典的。经典到什么程度呢？几乎任何讲深度强化学习的文章和书籍都会提到它。2013 版 DQN 各层的参数，如表 8-1 所示。

表 8–1　2013 版 DQN 各层的参数

Layer	Input	Filter Size	Stride	Num Filters	Activation	Output
Conv1	84 × 84 × 4	8 × 8	4	16	ReLU	20 × 20 × 16
Conv2	20 × 20 × 16	4 × 4	2	32	ReLU	9 × 9 × 32
FC3	9 × 9 × 32			256	ReLU	256
FC4	256			18	Linear	18

解释一下这个网络的结构。

网络的输入是一个连续 4 帧 84 × 84 的图像。也就是说，这个充当游戏机器人的网络，要在正式把样本输入网络之前，连续观测 4 帧图像，把这 4 帧图像连接在一起（可以理解为连成一幅“大图”），作为样本的状态 s 输入。为什么要使用连续的图像而不是一帧图像呢？原因很简单。这种游戏和象棋有一些差别。象棋的盘面是固定的，不论前面的棋子是怎么走的，到了当前的盘面后，盘面信息就能将状态完整地描述出来了。对一名成熟的棋手来说，只看当前的盘面就可以决定接下来走哪步棋最好了。当然，有人会说，前面的棋子是怎么走的，代表了对方的下棋风格（或者战略意图）。然而，在计算机面前，这些都不重要——任你什么风格，我只按照当前看到的盘面寻找最优解。

可是，Atari 游戏的输入通常是连续的，如图 8-1 所示。

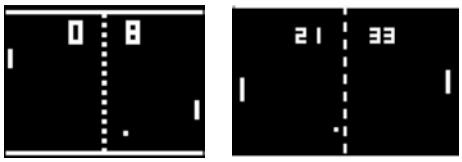


图 8-1　Atari 2600 乒乓球游戏

① 一个 ASR（Automatic Speech Recognition）模型，用来做语音识别。
② 一个 TTS（Text To Speech）模型，用来做语音合成。
③ 一个对抗生成网络的模型。

请尽量发挥你的想象力——是的，这是一个乒乓球游戏的界面，中间的虚线表示球网，两侧的白色线条表示球拍，在黑色背景上游走的白点表示乒乓球。如果只看静态图，你肯定无法判断当前乒乓球要向左移动还是向右移动，是自己刚刚打过去的还是对方打过来的——机器人就更没办法判断了。所以，如此重要的信息，要放在连续的多帧图像中传递。我们完全不用担心——因为网络有卷积核，可以用来做特征提取，而这些特征是可以通过训练得到的，所以，只要将信息输入网络，网络总会找出这些信息。

在第一层上，有 16 个 8×8 的卷积核，Stride 为 4。仅凭感觉来说，8×8 的卷积核属于特征提取粒度比较粗的情形，8×8 的区域会映射到一个点（这显然比 3×3 和 1×1 卷积核的提取粗糙一些）。Stride 为 4，即一次跳过 4 个像素，也说明作者在设计这个网络模型的时候就给输入的内容定了性——图像信息很稀疏。后面跟着一个 ReLU 函数，也就是把平面直角坐标系中的第二象限和第三象限的内容都转换成 0，过滤了其中的内容。输出是 16 个 20×20 的 Feature Map。接下来的一个卷积层，则对前面提取的内容信息进行了进一步的提取。

FC3 是一个全连接层。输入为 $9 \times 9 \times 32$，相当于 2592 维的向量；输出的是 256 维的向量。换句话说，FC3 层的待定系数就是一个 2592×256 的矩阵。

FC4 也是一个全连接层。它是一个 256×18 的矩阵，输入为 256 维的向量，输出为 18 维的向量。输出的 18 维都是什么呢？是 Atari 游戏机（如图 8-2 所示）的手柄输出的动作空间全集。

图 8-2　Atari 2600 游戏机

我们终于见到 Atari 2600 本尊了。主机就不介绍了，我们主要来看手柄。手柄上有一个八向的摇杆，还有一个按钮。两者组合起来，就是：

- 按钮：按下、不按，共 2 个状态。
- 摇杆：上、左、下、右、左上、左下、右上、右下、中，共 9 个状态。

这个手柄最多能输出 18 个（2×9）动作，因此，网络的最后一层输出的是 18 维的向量。

不过，论文里也没把话说得很绝对，只说由于游戏本身的需求，这个动作从 4 维到 18 维不等。例如，《吃豆子》那种游戏，确实只有 4 个动作可以输出（如图 8-3 所示）。

到此为止，网络的结构就介绍完了。仔细看看，没有发现任何特殊的地方，绝对是中规中矩的卷积层和全连接层结合的一个小网络，和用于实现 MNIST 分类任务的卷积神经网络很像。

那么，DQN 的特殊之处在哪里呢？在训练过程上。让我们一起往下看。

图 8-3　Atari 2600 PC-MAN 游戏

8.1.2　训练过程

DQN 的训练过程注定和 MNIST 不一样，因为如果一样，就是做分类了。具体怎么做，我们通过伪代码来了解，如图 8-4 所示。

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
　Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
　for $t = 1, T$ **do**
　　With probability ϵ select a random action a_t
　　otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
　　Execute action a_t in emulator and observe reward r_t and image x_{t+1}
　　Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
　　Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
　　Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
　　Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
　　Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
　end for
end for

图 8-4　2013 版 DQN 的伪代码

这段伪代码很短，算上标题还到 20 行，从结构上可以分成三层。

- 第一层：初始化一个 Replay Memory（称作“D”），其最大容量为 N。Replay Memory 是为了存储 Transition 信息而设置的，就是每个 Step 在玩游戏时收集的 (s, a, r, s')。然后，将随机数作为参数，初始化一个深度神经网络 Q。网络 Q 的参数为 θ。接下来，开启一个 M 次的循环，也就是做 M 个 Episode，让机器人在里面不停地玩游戏，直到通关或者失败。
- 第二层：在每一个 Episode 中初始化一个序列，然后进行预处理。在伪代码中没有对此做太多的解释，但实际上，这部分就是用来构造输入内容的（我们暂时不用细究）。在第二层中开启一个 T 次的循环，相当于这 T 次中的每一次都在处理一个 Step 的内容。
- 第三层：每个 Step 都在做什么呢？

用一个 ε 大小的概率去做一个随机动作 a_t。这个动作很容易理解，就是为了进行充分的试探。在这段伪代码中，虽然没有做过多的说明，但我们每个人都能想到，ε 是一个变化的值会比较合理。这个值在一开始比较大，就是为了让机器人充分地随机试探环境并得到反馈。越到后面，这个值应该越小——机器人的不断学习，会使其行为逐渐得到优化，变得越来越靠谱，所以，应该适当减小 ε 的值，使机器人不至于进化了那么久还会有相当的概率去做不靠谱的事情。

在设置 ε 概率之外要做什么呢？用 $1-\varepsilon$ 的概率计算

$$a_t = \max_a Q^*(\phi(s_t), a; \theta).$$

简言之，就是让当前的环境状态变量 s_t，通过当前的网络产生 18 个值。前面说过为什么会产生 18 个值，就是 18 个动作各自的评分。然后，将其中评分最高的动作作为输出到环境中的动作。这个动作相比一开始的动作，也非常不靠谱，跟随机没什么两样。不过，这个动作将逐渐进化，因为网络 Q 会逐步拟合 $s \to Q(s,a)$ 这个映射过程。这个拟合过程，和用神经网络拟合从一个输入到多维连续向量输出的过程没有区别——区别在后面。

从 Replay Memory *D* 中随机选择一些 Transition，进行如下计算：

$$y_i = \begin{cases} r_j \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) \end{cases}.$$

也就是说，y_i 有两种赋值的情况。

以上表达式中的前一种情况是，如果当前状态是游戏结束（也就是通关或者失败），y_i 就被赋值成当前得到的回报值。这里的回报赋值肯定是有讲究的。如果是通关，或者以通关为终极目标的任务，就一定要给出一个比较大的回报值（这相当于终极目标的实现）。而且，这个比较大的回报值会“逆流而上”，通过贝尔曼方程不断向前更新，把一棵树中的一条靠谱的路线标出来。如果失败了，那一定要赋值一个绝对值比较大的负数（至少我是这么想的）。这样，在使用贝尔曼方程进行更新的时候，只要有一条路可选，就永远不会有机会进行回溯了。

对其他情况，赋值为 $y_i = r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta)$。这种设置相当于，对在下一个状态下做的所有动作中的估值进行比较，拿出一个最大的作为“远期回报”，加上当前的回报 r_j，赋值给 y_i。这个赋值过程，就是在时间差分法的 Q-Learning 算法中用贝尔曼方程进行估值更新的过程。

Replay Memory 相当于一个池子，里面保存了一小段一小段 (s, a, r, s') 的 Transition 信息。在做估值更新时，会随机选择其中的一部分。这样做的好处是，能够消除这些样本在序列上的相关性，更高效地对 y_i 的值进行估算。Replay Memory 就像游戏世界中的“二十四史”，记录了曾经发生的各种 Transition 信息及历史评价 Reward，供后人研究。

在第三层的最后一步做优化，相当于优化损失函数

$$\text{Loss} = (y_i - Q(\phi_j, a; \theta))^2.$$

这还真是一个监督学习过程。标准的、平淡无奇的 MSE Loss，让这个 DQN 网络输入一个状态所

描述的 ϕ_j 产生 18 维的向量，让输出的向量值和进行价值评估回溯后的 y_i 不断靠近。因此，y_i 也应该是 18 维的向量。将向量和向量的对应维度，两两相减后算平方，就是损失函数的定义。

用简短的文字描述一下这个训练过程。在一个与 Q-Learning 极其类似的计算过程中，通过玩游戏收集 (s,a,r,s')，将其存入 Replay Memory。玩游戏过程中的每个 Step，都有一定的概率用随机动作去试探。而对其他情况，都是用目前的网络拟合出来的最优动作来做的。每个 Step 执行后，都要从 Replay Memory 里取出一些 (s,a,r,s') 的 Transition 信息来做估值更新，并把这些估值当成标签，用

$$\text{Loss} = (y_i - Q(\phi_j, a; \theta))^2$$

不断进行拟合，让 Q 网络根据这些估值来调整 $Q(s,a)$ 这个函数的估值。

这个过程是一个混合过程，我们来看两个极端的例子。一个是 Q-Learning 中用贝尔曼方程更新估值标签的过程，

$$y_i = \begin{cases} r_j \\ r_j + \gamma \max\limits_{a'} Q(\phi_{j+1}, a'; \theta) \end{cases};$$

另一个是用损失函数

$$\text{Loss} = (y_i - Q(\phi_j, a; \theta))^2$$

进行监督学习的过程。

如果只做标签更新，那就是纯粹的 Q-Learning；如果在标签值（动作估值）准确的情况下，做以损失函数为约束的梯度下降，那就是监督学习。现在是两者混在一起做。你看是不是这样？直观地看，似乎确实如此——希望通过每一次每个 Step 学到的有限且不准确的动作估值来选择动作，然后一点点逼近我们理想中的 $Q(s,a)$ 的过程。

DQN 2013 这个模型比较旧。要想全面认识 DQN 算法族，我们还需要认识它的下一个版本，也就是 DQN 2015（参见 8.2 节）。

8.1.3　Replay Memory

DQN 中使用了一种叫作“Replay Memory”的组件，也就是我们在 8.1.3 节中看到的那个叫作“*D*”的组件。它通常用来存放收集到的 (s,a,r,s')。

为什么要设置 Replay Memory，并且，要伴随每个 Step，从里面拿出一些 Transition 来做估值更新呢？我们在标准的 Q-Learning 算法中是没有见过 Replay Memory 的，既然将它引入，那么它必然有自己的优势。总的来说，它的优势就是打破了 (s,a,r,s') 序列的相关性。在这里，就要解释一下标准的 Q-Learning 没有打破序列相关性所带来的问题。

在使用动态规划法、蒙特卡罗法及时间差分法分别对一个估值进行更新时，我们会发现，蒙特卡罗法有一个很大的问题：因为它每次都是从一个 Episode 开始，一直走到这个 Episode 的

Terminal 状态，才进行奖励值回溯的，所以，这条路上的奖励值的估值仅来自某个策略的某一次行动。总结一下，问题就是：估算会存在偏颇。

如果不打乱顺序，每次回溯时都按照邻近的样本序列来更新，就会使每次估算都比较“偏”，更新时估值的方差过大。方差越大，收敛的效果越差。也就是说，对每个状态 s 的每个动作 a 的估值，每次估算的差距都非常大。

8.1.4 小结

2013 版 DQN 结构简单，效果也比较理想。论文中给出了一些参考数据，如图 8-5 所示。左列中是不同的算法模型，其他列中是不同的游戏类别。

	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Random	354	1.2	0	−20.4	157	110	179
Sarsa [3]	996	5.2	129	−19	614	665	271
Contingency [4]	1743	6	159	−17	960	723	268
DQN	**4092**	**168**	**470**	**20**	**1952**	**1705**	**581**
Human	7456	31	368	−3	18900	28010	3690
HNeat Best [8]	3616	52	106	19	1800	920	**1720**
HNeat Pixel [8]	1332	4	91	−16	1325	800	1145
DQN Best	**5184**	**225**	**661**	**21**	**4500**	**1740**	1075

图 8-5 2013 版 DQN 参考数据

可以看出，尽管 DQN 在某些游戏中的表现强于人类，但数量太少了。值得一提的是，DQN 的表现比其他人工智能算法强得多。

读完对 2013 版 DQN 的介绍后，相信读者朋友已经有了一个认识——DQN 还是挺“弱”的。好在，研究从未止步。

8.2 2015 版 DQN

2015 年，DeepMind 对 2013 版 DQN 进行了一次重大的改造，并在 *Nature*[①]上披露了这个模型的表现。我个人认为，这次改造对深度强化学习领域的研究有比较深远影响——把 2013 版 DQN 从一个网络变成了由两个网络组成的模型，出现了主网络（Main Network）和目标网络（Target Network）的概念。

8.2.1 模型结构

2015 版 DQN，从原始刊载于 *Nature* 上的内容来看，在模型上没有太多新意，只提供了一幅网络示意图，如图 8-6 所示。

① 著名科技期刊《自然》。

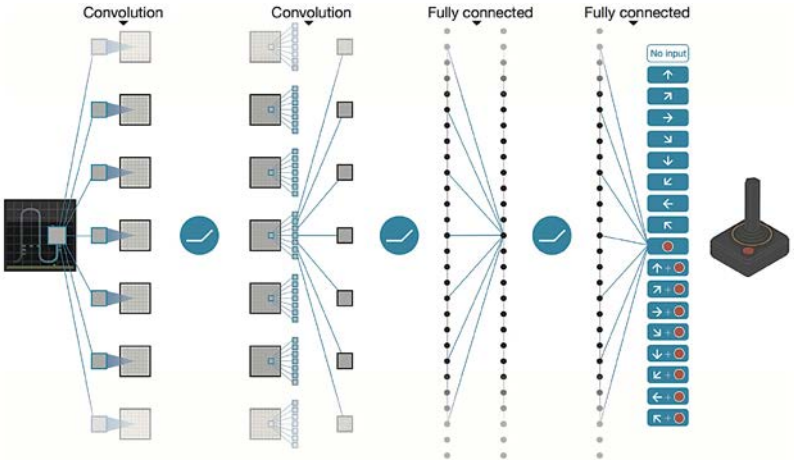


图 8-6　2015 版 DQN 的网络示意图

文章里没有提到这个网络的每一层中究竟有哪些参数，只能看出前面有两个卷积层，后面有两个全连接层。从结构上看，和 2013 版 DQN 没有明显的差别。其实，这不重要，因为一个神经网络多一个卷积层或者少一个卷积层、多一个卷积核或者少一个卷积核，在大多数情况下不会对网络的训练产生本质的影响（充其量是更容易引起过拟合或者欠拟合）。重要的、使得 DQN 产生本质变化的方面，就是主网络和目标网络的分离。所以，我们不用为具体的参数取值而纠结。

8.2.2　训练过程

和认识 2013 版 DQN 时一样，我们还是先看看 2015 版 DQN 是怎么训练的。2015 版 DQN 的伪代码，如图 8-7 所示。

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $\left(\phi_t, a_t, r_t, \phi_{t+1}\right)$ in D
    Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

图 8-7　2015 版 DQN 的伪代码

一眼就能看出，2015 版 DQN 的伪代码在结构上与 2013 版 DQN 的伪代码非常相近。

- 第一层：初始化一个 Replay Memory，称为“*D*”，大小为 *N*。初始化一个 Action-Value 网络 $Q(s,a|\theta)$ 和一个 Target Action-Value 网络 $\hat{Q}(s,a|\theta^-)$。不仅两个网络的结构一模一样，在初始化时，两个网络的参数也一模一样。然后，开启一个 *M* 次的循环，也就是做 *M* 个 Episode，让机器人在里面不停地玩游戏，直到这个 Episode 结束为止。
- 第二层：在一个 Episode 内初始化一个 *T* 次的循环，每次循环对应于一个 Step。
- 第三层：首先，使用概率 ε 去做一个随机的动作 a_t。然后，使用概率 $1-\varepsilon$ 计算 $a_t=\max\limits_a Q(\phi(s_t),a;\theta)$。注意这个地方，在形式上和 2013 版 DQN 非常像。在这里，真正输出动作的网络是 Action-Value 网络。

收集本次产生的 Transition 信息，进行 y_i 值的设置：

$$y_i=\begin{cases} r_j \\ r_j+\gamma\max\limits_{a'}Q\left(\phi_{j+1},a';\theta^-\right) \end{cases}.$$

发现区别了吗？这时的一个状态 s 的估值 y_i，并不是由 Action-Value 网络给出的，而是由 Target Action-Value 网络给出的。

损失函数的优化过程是一样的，优化的是 Action-Value 网络：

$$\text{Loss}=(y_i-Q(\phi_j,a;\theta))^2.$$

最后，是一个类似触发器（或者说计数器）的东西。每隔 *C* 步，设置 $\hat{Q}=Q$，即将 Target Action-Value 网络的参数赋值为 Action-Value 网络的值。

模型就介绍到这里，示例代码请访问本书的 GitHub 页面下载。

8.2.3 Target 网络

2015 版 DQN 的其他内容，与 2013 版 DQN 相比，只多了一个 Target 网络。Target 网络在这里只提供了一个估值 y_i，供主网络作为标签值去学习。在这个过程中，每隔 *C* 步，就把 Target 网络的参数赋值为主网络的参数——这就是全部内容。

我们思考一下：如果 *C* 的值为 1，是什么样的情况呢？*C* 的值为 1，也就是说，每一步都要更新一次，那么，Target 网络就完全没必要存在了，直接使用主网络就可以了（如图 8-8 所示）。因此，根据直觉我们就能判断：这个值要设置得稍微大一些。

于是，这个反应“迟钝”一些的 Target 网络给我们带来的好处也比较明确了。用主网络进行估值，每一次 y_i 的值都会发生变化——一个频繁变化的标签是会让神经网络“抓狂”的。就像向 MNIST 数据集中输入一幅图片后，一会儿告诉神经网络它是 0，一会儿告诉神经网络它是 2，神经网络该怎么办？

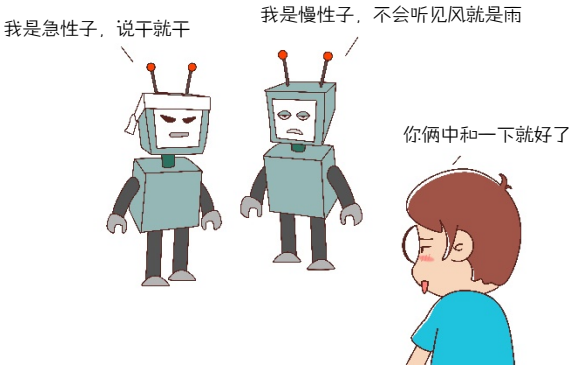


图 8-8　主网络机器人和目标网络机器人

标签不断变化会带来什么问题？标签不断变化，将直接导致损失函数的“形状”——第 6 章中说到的那只“碗”——不断变化。“碗”本身就有边缘和褶皱，这些褶皱还在不断变化，因此，网络的收敛性将很难得到保证（生成对抗网络中也有类似的问题）。Target 网络的好处是：可以在一段时间内使拟合的标签相对稳定，从而让主网络的收敛平稳一些。

由于这种结构被验证为“非常有效”，可以让神经网络的训练平稳收敛，便拉开了广泛使用 Target 网络的序幕。后来出现的所有深度强化学习模型，几乎无一例外，都使用了 Target 网络。

8.2.4　小结

2015 版 DQN 比 2013 版 DQN 更为健壮。虽说在网络拓扑结构层面没有变化，但在整体训练的格局（或者说架构）上有非常大的突破，即引入了一个可以帮助主网络稳定训练的目标值的 Target 网络。应该说，这是学术界公认的最早的使用深度学习技术解决强化学习问题的手段。这两个网络，也分别由其最初发表的刊物（会议）名称命名，2013 版 DQN 也称“NIPS DQN”，2015 版 DQN 也称“Nature DQN”。

这些进步看上去小小的，但事实上一点都不“小”。作为深度强化学习大厦的基石，越来越多的 Trick 被引入，并逐渐成为训练中不可或缺的步骤。

当然，DQN 的发展远远没有结束。其多个变种，用各种各样的新方法，实现了更多、更有效的训练方式。下面，我们一起了解一下其中最著名的两个模型——Double DQN 和 Dueling DQN。

8.3　Double DQN

Double DQN 的概念，也是由 DeepMind 团队提出的，论文名为 *Deep Reinforcement Learning with Double Q-Learning*，发表于 2015 年。这篇论文的前身是 2010 年发表在 NIPS 上的论文 *Double Q-Learning*。从题目上看，后者讨论了如何用 Double Q-Learning 模型做强化学习训练，前者则把这个思想迁移到了深度强化学习领域。

8.3.1 模型结构

Double DQN 模型的思路是从 Double Q-Learning 迁移而来的，也就是从传统机器学习迁移而来的，因此，不存在必须有一个神经网络结构才能训练的问题。也就是说，只要有两个一模一样的 Q-Learning 模型或者两个 DQN 网络，就可以构成“Double”模型，而剩下的就是训练了。

需要注意的是，Double Q-Learning 是为了解决过估计问题而设计的。过估计问题在第 5 章中已经介绍了，关键在 Max 的取值上。按理说，一个状态的估值应该是这个状态下所有动作估值的数学期望值（也就是加权平均值）才对，可是，如果每次都用 max 取最大值，这个最大值和加权平均值之间的差距就形成了误差——这一定会导致过估计的发生。

8.3.2 训练过程

我们照例来看伪代码，如图 8-9 所示。

Algorithm 1 Double Q-learning

Initialize Q^A,Q^B,s
repeat
  Choose a, based on $Q^A(s,\cdot)$ and $Q^B(s,\cdot)$, observe r, s'
  Choose (e.g. random) either UPDATE(A) or UPDATE(B)
  if UPDATE(A) **then**
    Define $a^* = \arg\max_a Q^A(s',a)$
    $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)\left(r + \gamma Q^B(s',a^*) - Q^A(s,a)\right)$
  else if UPDATE(B) **then**
    Define $b^* = \arg\max_a Q^B(s',a)$
    $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)(r + \gamma Q^A(s',b^*) - Q^B(s,a))$
  end if
  $s \leftarrow s'$
until end

图 8-9 Double Q-Learning 的伪代码

也是一段短小的伪代码。首先，初始化两个网络 Q^A 和 Q^B，以及一个状态 s。然后，通过一个循环，用 Q^A 和 Q^B 交替进行估值计算和动作输出。把状态 s 分别输入 Q^A 和 Q^B，必然会得到两个网络分别输出的动作 a，并且会根据各自的动作迁移到各自的状态 s'，得到各自的奖励值 r。接下来，随机选择网络 Q^A 或者 Q^B（应该以等同概率选择），对其进行更新。

如果选择更新 Q^A，我们称为做了 UPDATE(A) 操作。此时，在使用 $Q^A(s',a)$ 输出一个状态 s' 后的众多动作 a 中，将估值最大的那一个作为 $a*$。使用 $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)(r + \gamma Q^B(s',a*) - Q^A(s,a))$ 的方法进行更新，即使用 Q^B 网络对刚刚 Q^A 网络“认为”最有价值的动作 $a*$ 进行估值，以 Q^B 的估值作为学习目标来做更新操作。可以看出，在整个更新式中，除了 Q^B 处的上标有变化，其他的值和普通的 DQN 没有本质区别。

如果选择更新 Q^B，我们称为做了 UPDATE(B) 操作。UPDATE(A) 与 UPDATE(B) 是一套除名字做了交换以外没有任何区别的对称动作。

此时，在使用 $Q^B(s',a)$ 输出一个状态 s' 后的众多动作 b 中，我们将估值最大的那一个作为 $b*$。以 $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)(r + \gamma Q^A(s',b*) - Q^B(s,a))$ 的方法进行更新，即使用

Q^A 网络对刚刚 Q^B 网络“认为”最有价值的动作 $b*$ 进行估值，以 Q^A 的估值作为学习目标来做更新。

$Q(s,a)$ 的目标一直在变化（如图 8-10 所示）。

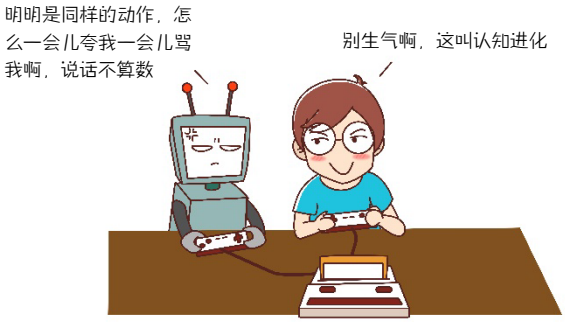


图 8-10　$Q(s,a)$ 的目标一直在变化

Double DQN（或者说 Double Q-Learning）带来的好处，就是在学习估值的时候用 max 来做评估，在选择输出动作的时候用另一个没有做 max 评估的权重集合来做动作，从而在一定程度上降低了发生过估计的可能性。也可以说，这种用两个网络进行动作选择和动作估值解耦的手段，可以在一定程度上避免过估计正噪声的积累。

内容并不复杂，示例代码请访问本书的 GitHub 页面下载。

8.3.3　效果

在 *Deep Reinforcement Learning with Double Q-Learning* 一文（就是提出 Double DQN 的那篇论文）中，作者专门对实验结果进行了量化，如图 8-11 所示（见彩插）。

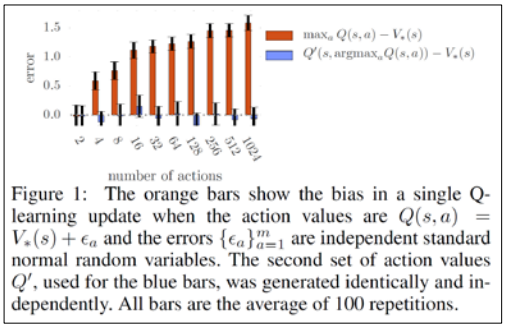


Figure 1: The orange bars show the bias in a single Q-learning update when the action values are $Q(s,a) = V_*(s) + \epsilon_a$ and the errors $\{\epsilon_a\}_{a=1}^m$ are independent standard normal random variables. The second set of action values Q', used for the blue bars, was generated identically and independently. All bars are the average of 100 repetitions.

图 8-11　Double DQN 与 DQN 在过估计方面的对比

横坐标表示动作空间的数量，左侧是仅有 2 个动作的情况，右侧是 1024 个动作的情况。纵坐标表示估计值的对比：较长的橙色竖条，表示 DQN 中估值最大的动作 a 的估值与标准状态值 s

的差，即 $\max_a Q(s,a) - V_*(s)$；较短的蓝色竖条，表示通过 Double DQN 的方法得到的估值，也就是 $Q'(s, \arg\max_a Q(s,a)) - V_*(s)$。与标准值相比，Double DQN 的误差非常小——至少比 DQN 小很多。而且，DQN 产生的过估计的严重程度，会随着动作空间的增大而提高，但 Double DQN 不会导致这样的问题。

此外，可以参考论文中的一组数据，如图 8-12 所示。

	DQN	Double DQN	Double DQN (tuned)
Median	47.5%	88.4%	116.7%
Mean	122.0%	273.1%	475.2%

Table 2: Summary of normalized performance up to 30 minutes of play on 49 games with human starts. Results for DQN are from Nair et al. (2015).

图 8-12 DQN 与 Double DQN 得分能力的对比

这是 DQN 和 Double DQN 得分能力的对比。“Median”表示中位数，“Mean”表示平均数。在 49 个游戏中，让 DQN、Double DQN 和 2015 版 DQN 比较得分情况，可以看出，差别还是挺大的。不管是经过调优（Tuned）的 Double DQN，还是“原生态”的 Double DQN，与 2015 版 DQN 相比，“成绩”都有大幅提高。

这个百分比的计算公式为：

$$\text{score}_{\text{normailzed}} = \frac{\text{score}_{\text{agent}} - \text{score}_{\text{random}}}{\text{score}_{\text{human}} - \text{score}_{\text{random}}}.$$

这是一个经过归一化处理的比值，即用算法玩游戏的得分减去随机玩游戏的得分，除以人类玩游戏的得分减去随机玩游戏的得分。

8.3.4 小结

过估计问题是 Q-Learning 算法与生俱来的，并遗传给了“子孙后代”（DQN 的各个变种）。

从统计学的角度看，如果在输出端对每个动作的估值 $Q(s,a)$ 的过估计幅度都是一样的，也就是说，如果所有的动作都被过估计了，而且过估计的幅度相同，那么，最终选择动作时，仍然与没有被过估计时一样。因此，这样处理不会影响最终的策略。这很容易理解，相当于将 $Q(s,a)$ 的每个 a 的估值等比例扩大或者缩小，而这不会影响它们的排名。

如果过估计是不均匀的（也就是说，会影响排名），就比较麻烦了。在这种情况下，很可能会学到一种次优解（或者说策略）。这时，Double Q-Learning 就能发挥作用了。

8.4 Dueling DQN

在众多 DQN 变种中，有一种叫作“Dueling DQN”的模型。

该模型是在 2016 年发表的论文 *Dueling Network Architectures for Deep Reinforcement Learning*

中被提出的，它也是 DeepMind 的大作之一。“Dueling”的英文意思是“使某人参加决斗”。在这篇论文的摘要中有这么一句：“Our dueling network represents two separate estimators: one for the state value function and one for the state-dependent action advantage function.”大意是：“该 Dueling 网络的形式为，提供两个分离的估值函数，一个是状态估值函数，另一个是独立于状态的动作优势估值函数。”这就是它的本意。至于要不要跟“决斗”产生关系，我觉得没有必要。

8.4.1　模型结构

Dueling DQN 的结构比较有趣，是一个中间分叉的结构。DQN 和 Dueling DQN 的结构对比，其中每个白色的立方体都代表一层，如图 8-13 所示（见彩插）。

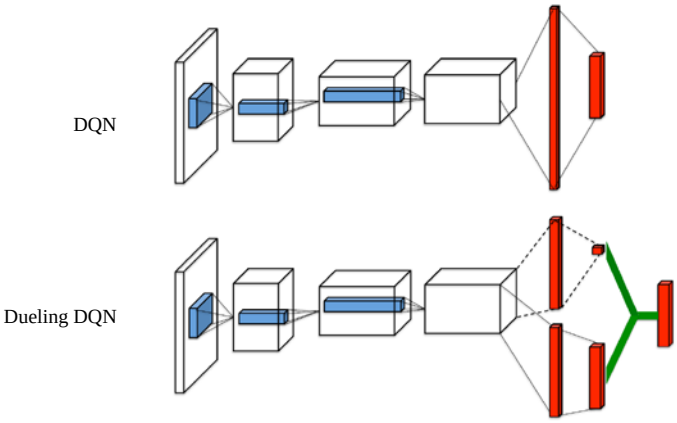


图 8-13　DQN 和 Dueling DQN 的结构对比

在如图 8-13 所示的 DQN 中，最后一层中有一长一短两个条状向量。这两个条状向量将前面三个卷积层输出的内容“拉直”了，其中短一些的向量表示最终输出的 Action（例如，Atari 2600 游戏机的那 18 个动作）。这两层之间使用全连接网络。

如图 8-13 所示的 Dueling DQN 与 DQN 相比，只做了很少的修改。被“拉直”的条状向量出现了两个分支。上面的分支映射到一个叫作 $V^{\pi}(s)$ 的输出，这就是状态估值函数，用“白话”说，就是估算一个状态本身有多少价值。下面的分支映射到 $A^{\pi}(s,a)$，这就是所谓独立于状态的动作优势估值函数。从形式上看，这也是在计算一个状态 s 下的 a 的估值。那么，这个结构所表示的含义和原来的 DQN 究竟有什么不同？假设有这样一个前提：

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s).$$

也就是说，一个完整的估值函数 $Q^{\pi}(s,a)$（就是前面一直讨论的 Q 函数），实际上是由两部分组成的，一部分是由状态 s 下的动作质量 a 贡献的，另一部分是由状态 s 本身的价值决定的，两部分直接线性叠加，构成了 $Q^{\pi}(s,a)$。对这个公式做简单的移项，就能得到

$$A^{\pi}(s,a)+V^{\pi}(s)=Q^{\pi}(s,a),$$

网络相当于对这样一个假设直接进行了相应的实现。最后一层就是两个值直接相加得到的输出 $Q^{\pi}(s,a)$。这种包含两个分支的网络，前面的网络是共享的，后面的网络则各有各的待定系数。

这两个分支网络真的能分别学到自己应该学到的东西吗？答案是肯定的。例如：当前的情况非常好，不管做什么动作，都能在游戏中“活”得很好；当前的情况非常糟糕，不管做什么动作，都会“死”在游戏中。在传统的 DQN 模型中，在这两种情况下，s 通过特征提取产生的特征肯定是一样的，最后一层的 $Q(s,a)$ 的输出动作 a 则完全不同，而 Q 的估值是一样的。

为了说明这种模型的意义，论文给出了一幅图，如图 8-14 所示（见彩插）。上面这组图表示，路上车很少、做什么动作都可以的情况；下面这组图表示，路上车很多、动作选择很重要的情况。被染为红色的部分说明：在上面这组图表示的情况下，状态估值函数会观察当前的状态（主要是观察车道的情况），而动作优势函数就没什么反应了（因为当前什么都不做就已经很好了）；在下面这组图表示的情况下，不仅状态估值函数注意到了这种比较复杂（或者说比较糟糕）的状态，动作优势函数也注意到正前方有车辆需要避开。

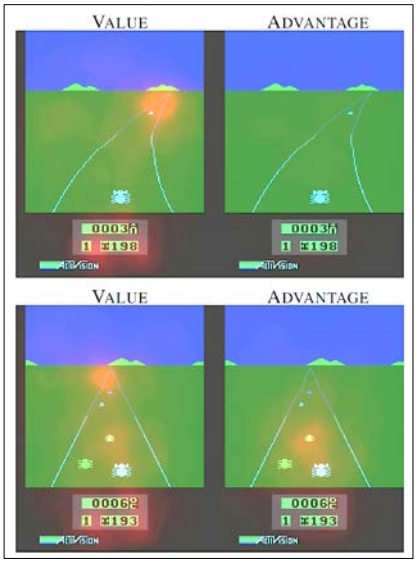


图 8-14　Dueling DQN 的状态估值和动作优势估值的比较

这种方式还是有一定意义的，如图 8-15 所示。由于从状态 s 到 $Q^{\pi}(s,a)$ 的上下两段是分开的，所以，如果对同一个状态 s，不同的 a 得到了相同的 $Q^{\pi}(s,a)$ 奖励值，势必使 $A^{\pi}(s,a)$ 对 a 的变化不敏感，这部分网络的梯度相对较小，更新也相对不积极。反之，如果同一个状态 s 做不同的动作 a，得到了奖励值截然不同的 $Q^{\pi}(s,a)$，就会使 $V^{\pi}(s)$ 对状态不感兴趣，由 $A^{\pi}(s,a)$ 贡献主要的成分。

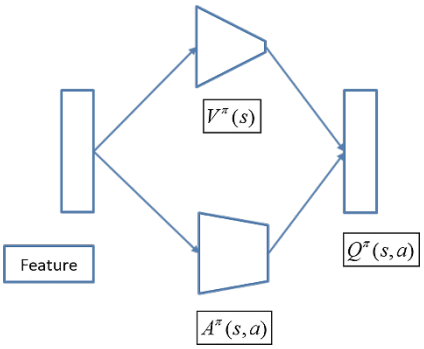


图 8-15　Dueling DQN 的 $A^{\pi}(s,a)$ 和 $V^{\pi}(s)$

在最终的实践中，对于 $A^{\pi}(s,a)+V^{\pi}(s)=Q^{\pi}(s,a)$，可以有如下变化：

$$Q(s,a;\theta,\alpha,\beta)=V(s;\theta,\beta)+(A(s,a;\theta,\alpha)-\max_{a'\in|A|}A(s,a';\theta,\alpha))$$

或

$$Q(s,a;\theta,\alpha,\beta)=V(s;\theta,\beta)+(A(s,a;\theta,\alpha)-\frac{1}{|A|}\sum_{a'}A(s,a';\theta,\alpha)).$$

这个变化的含义是：两个小网络共享部分的参数是前面卷积神经网络的 θ，β 是它自己所拥有的这部分网络的参数，所以，完整的状态估值函数应该写作 $V(s;\theta,\beta)$；同理，把下面的网络写成 $A(s,a;\theta,\alpha)$。

后面为什么要减去一项呢？因为 $Q(s,a;\theta,\alpha,\beta)$ 只是一个估值，其中的两个分量 $V(s;\theta,\beta)$ 和 $A(s,a;\theta,\alpha)$ 不一定准确。例如，同时给这两个分量分别加上/减去一个浮点数，最终的 Q 不变，此时，分别对 $V(s;\theta,\beta)$ 和 $A(s,a;\theta,\alpha)$ 进行拟合，却得不到本来应该得到的估值意义（或者说，发生了过拟合）。因此，论文作者强制指定优势函数在选择动作时的优势值为 0，也就是说，$A(s,a;\theta,\alpha)-\max_{a'\in|A|}A(s,a';\theta,\alpha)$ 这一项等于 0，此时自然有 $Q(s,a;\theta,\alpha,\beta)=V(s;\theta,\beta)$，由此得到的 $V(s;\theta,\beta)$ 为价值函数的估值，$A(s,a;\theta,\alpha)$ 为动作优势函数的估值。

用 $-\frac{1}{|A|}\sum_{a'}A(s,a';\theta,\alpha)$ 取代 $-\max_{a'\in|A|}A(s,a';\theta,\alpha)$ 也是一种可行的方式，就是减去该状态下所有动作的优势值的均值。这两种方法的目的是一样的。

Dueling DQN 和普通 DQN 的训练过程没有差异，在这里就不单独展开讲解了。相关代码请访问本书的 GitHub 页面下载。

8.4.2　效果

在 *Dueling Network Architectures for Deep Reinforcement Learning* 这篇论文中，作者对 Dueling DQN 和 2015 版 DQN 进行了对比（其中的百分比是做过归一化的）：

$$\frac{\text{Score}_{\text{Agent}} - \text{Score}_{\text{Baseline}}}{\max\{\text{Score}_{\text{Human}}, \text{Score}_{\text{Baseline}}\} - \text{Score}_{\text{Random}}}.$$

“Agent”表示 Dueling DQN 模型，“Baseline”表示 2015 版 DQN，“Human”表示人类的水平，“Random”表示随机“瞎打”的结果。

如图 8-16 所示，除了少数几个游戏，对绝大部分游戏使用 Dueling DQN，都产生了更好的效果——无论是和人类相比，还是和 2015 版 DQN 相比。

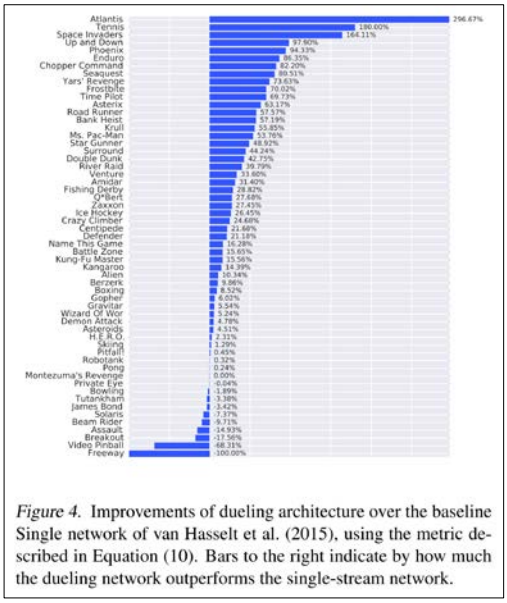


Figure 4. Improvements of dueling architecture over the baseline Single network of van Hasselt et al. (2015), using the metric described in Equation (10). Bars to the right indicate by how much the dueling network outperforms the single-stream network.

图 8-16　Dueling DQN 的实际表现对比

8.4.3　小结

Dueling DQN 使用了一种比较巧妙的方式，对状态的估值和独立于状态的动作优势估值进行了各自独立的价值评估，使得它能够将动作本身的价值和状态本身的价值解耦。

怎么理解比较简单呢？例如，在评估下棋的策略时——有的盘面本身价值很高，也就是说，眼看就要赢了，走哪一步都差不多；有的盘面价值一般，状态胶着，局势不明朗，但是一些步子走出来价值就很高，步子和步子之间的估值差异比较大。这种解耦的假设，也会帮助网络学到更为准确的状态价值的估值，从而更有效地找到好的策略。

8.5　优先回放 DQN

在本节中，我们来认识一个叫作“优先回放”的概念。需要强调的是，优先回放这个概念出

现得比较晚，是在 2016 年发表于 ICLR[①]的论文 *Priorized Experience Replay* 中被提出的。优先回放的概念理解起来并不难，它既不针对 DQN，也不排斥传统 Q-Learning 算法。

优先回放是一种可以让 Q-Learning 学习效率更高的方法。它解决的就是传统 Q-Learning 算法收敛效率低的问题。问题出在哪儿呢？出在贝尔曼方程上。

还是看这个更新表达式：

$$Q(S,A) \leftarrow Q(S,A) + \alpha \left(R + \gamma \max_{a'} Q(S',a) - Q(S,A) \right).$$

虽然表达式本身没有问题，但 $Q(S,A)$ 的更新量就是 $\alpha(R + \gamma \max_{a'} Q(S',a) - Q(S,A))$ 的大小。想想看，$\alpha(R + \gamma \max_{a'} Q(S',a) - Q(S,A))$ 有没有可能为 0，也就是说，TD-error 有没有可能为 0？如果为 0，这种更新的计算还有没有意义？

首先，绝对有可能为 0。当奖励值的回溯比较彻底，整棵树已经无法收敛的时候，也就是 $Q(S,A)$ 和 $R + \gamma \max_{a'} Q(S',a)$ 的大小一样的时候，这个更新值完全可能为 0。你能想象在什么时候会这样吗？例如，在已经进行了一次这样的更新后，再进行一次更新，就会出现这种现象。那么，对于这种已经进行了充分回溯的树权，我们就没有必要再进行回溯了，应该优先挑选其他树权进行回溯。

回溯过程就是一个推理与学习的过程。对已经学好、学完的东西，就不需要再学一遍了，还是把机会留给那些估算差距较大的树权好了（因为对差距越大的部分，越应该优先进行回溯）。因此，每次从 Memory Buffer 里挑出来的 Transition 通常不是等权重的，TD-error 较大的 Transition 会被优先挑选出来进行更新。

因为优先回放不算一个新的模型算法，所以在这里就不做详细的分析和说明了。至于代码，请参考本书的 GitHub 页面。

8.6　本章小结

DQN 算法族是一个非常有代表性的算法族。可以说，它既从根本思路上继承了经典强化学习理论中 Q-Learning 算法的精髓，又融入了深度神经网络的高级特性，所以，自问世以来一直被人们所关注（如图 8-17 所示）。

在每年的新发表论文里，仍然包含大量的 DQN 新变种。这些变种都期望通过给原来的 DQN 加上各种各样的新的限制条件，让 DQN 向稳定、快速收敛的方向“进化”。因此，DQN 在未来仍然会是一个热门的方向。我们能在网上看到各种 DQN 模型，例如 Gorila DQN、Vanilla DQN，且每年都会推陈出新。

① ICLR（International Conference on Learning Representations），机器学习领域的顶级会议。

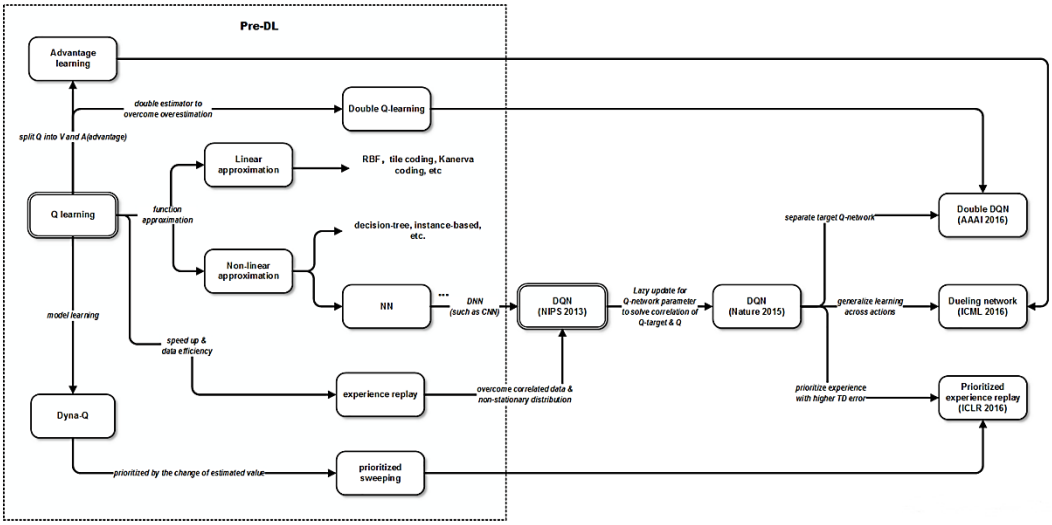


图 8-17 传统强化学习算法和 DQN 算法族的“进化”

在这里，我只是想提醒大家，DQN（或者说，其所基于的 Q-Learning）有不少先天缺陷。例如，过估计问题（在第 5 章中介绍过）。再如，基于 $S \rightarrow A \rightarrow R \rightarrow S'$ 这样的 Transition 本身的局限性，在一个高维输入环境中，如果想让来自不同的 Episode 的 Transition 通过贝尔曼方程的更新公式

$$Q(S,A) \leftarrow Q(S,A) + \alpha(R + \gamma \max_{a'} Q(S',a) - Q(S,A))$$

不断进行奖励值回溯，简直难到家了。在这种情况下，不仅 $Q(S,A)$ 中的动作 A 要足够丰富，试探出来的 S' 也要足够丰富，才有可能得到比较好的回溯结果。毕竟只通过一次试探得到的结果，相当于对一个盘面只进行一次尝试，那么，其他尝试中的更好的解释就得不到了。因此，结论令人窒息——对于那种又“高”又“茂盛”的树（希望你能明白我在说什么），需要“地老天荒”般的训练时间。此外，用于存储 Transition 的 Memory Buffer 也要非常大，大到单台计算机承载起来十分吃力的程度。

第 9 章　PG 算法族

在第 8 章中，我们一起认识了 DQN 算法族，这些算法都是以求解一个状态或者一个状态下某个动作的估值为手段的“间接”求解策略的方法。而在本章中，我们要接触一类与 DQN 算法族截然不同的方法，那就是“直接”求解策略的方法。

在使用间接的方法求出 $Q(s,a)$ 的值，形成完整的列表之后，我们得到的并不是一个完整的策略，而是一个列表。机器人输入一个状态 s，相当于将这个 s 作为索引，遍历 $Q(s,a)$，通过比较得到一个估值比较高的 a。这已经够“间接”了——每次都要通过查表获得一个动作，然后将一个个片段连接起来，才能构成一个完整的、貌似思维连贯的策略。

那么，有没有能够直接让一个模型或者网络“学”到一个策略的方法呢？也就是说，使一个输入的 s 直接对应于一个动作而不用再查表呢？其实是有的，就是策略梯度法（Policy Gradient），简称“PG 方法”。

9.1　策略梯度

策略梯度法其实对应于 2.4.1 节介绍的直接法。怎么理解比较好呢？让我们对比 2.4.2 节介绍的间接法来理解。

间接法希望计算出每个状态的价值 $V(s)$，以及每个状态下每个动作的估值 $Q(s,a)$，然后根据类似查表的方式（或者使用任何其他算法）来实现这样一个函数。策略就是每个动作都输出估值最大的 a。这个想法也是比较自然的，Q-Learning 及 DQN 算法族大都是为了实现这样一个目标而不断努力的。

PG 方法则不同，下面我们就讨论一下这种“直接法”的思路。

既然最终策略的形式是 $a=\pi(s|\theta)$，我们就应该定一个目标，即如何评价一个策略是好的还是不好的。不论是一个好的策略，还是一个不好的策略，都是以

$$a=\pi(s|\theta)$$

为表现形式的策略。将拥有这样一个策略的机器人放到某个环境中，每次都能得到一条“路径”。这个所谓“路径”，就是在策略 $a=\pi(s|\theta)$ 的引导下，一步步形成的一连串

$$s \to a \to r \to s.$$

毫无疑问，在这种情况下，好的机器人应该具备两个比较明显的素养。

一是要走得远。根据直观的感受，走得越远的机器人的策略应该越优秀。

二是要走得好。什么叫“好”呢？最基本的评价方式就是策略能够保证得到的奖励值尽可能大。只要目标明确且能够量化，剩下的就是比较量化值大小的问题了。

在这里，我觉得还是有必要强调一下设定奖励值的意义。作为一个重要的超参数，奖励值能够直接体现“价值观”，所以，对有重大而深远影响的行为一定要有“高额”奖励，反之应该惩罚。这样是不是就能描述一个有关策略评价的函数呢？应该是可以的。

假设策略 π 会引导机器人经历一个状态序列

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_t, a_t, r_t),$$

τ（希腊字母，读音为“套”）表示一个状态、动作、奖励值不断迁移的过程。

再设一个评价函数

$$J(\theta) = E[r_0 + r_1 + r_2 + \ldots + r_t | \pi_\theta].$$

从形式上看，这个评价函数符合数学期望值的形式。方括号里面的部分表示，在以 θ 为参数的策略 π_θ 的条件下，会产生一系列奖励值 $r_0 + r_1 + r_2 + \ldots + r_t$。需要注意的是，在策略 π_θ 确定的情况下，机器人每次玩游戏时走的不一定是完全相同的路线。因为环境这个东西是很复杂的：开始的状态就可能不一样；中间状态转移的概率我们也不清楚；每次能走的步数也不是确定的，可能有的 Episode 走得远，能直接“通关”，而有的 Episode“点儿背”，没走几步就“挂”了。所以，要想对一个策略进行公平、准确的估值，就要取多个策略的平均值，也就是前面的 E（数学期望值）。π_θ 可以直接理解成一个神经网络——反正没有规定在这里不能使用神经网络。

然后呢？对神经网络这东西，我们已经见识过通过梯度下降求损失函数极小值的过程了（参见第 6 章）。通过正向传播产生拟合值，与标签值做“差”计算，产生误差值，通过误差值求和产生损失函数，对损失函数用梯度下降法求极小值。而优化的对象，就是整个神经网络的参数 θ。现在的情况似乎比求损失函数极小值的情况“直接”一些。正向传播产生动作，动作在环境中产生回报值，通过回报值求和产生评价函数——难道我们不能对评价函数做“梯度上升”吗？能求极小值，就能求极大值。误差能最小化，得分就能最大化。接下来，就应该是优化网络了。参数的梯度能求出来吗？当然可以，往下看。

尝试将 $J(\theta) = E[r_0 + r_1 + r_2 + \ldots + r_t | \pi_\theta]$ 函数改写成

$$J(\theta) = E_{\tau \sim \pi_\theta}[\sum_t \tau] = \frac{1}{N} \sum_i \sum_t r(s_{i,t}, a_{i,t})$$

的形式，也就是将两个∑套在一起。外面的一层表示实验要做好多次，在这里就是下标 i；里面的一层表示一个 Episode 要走到底，在这里用下标 t 表示；$\frac{1}{N}$ 表示经历了 N 个状态，计算状态的平均得分是多少。就像在评价一位股票投资人的赚钱能力时，会使用类似平均年盈利能力或者平均月盈利能力的指标——盈利 50% 是 3 年实现的还是 1 年实现的，差得可远了。$r(s_{i,t}, a_{i,t})$ 就是指每一次获得的奖励值，只不过，为了区分单个点的奖励值，在这里都是基于多个不同 Episode

的不同状态的不同动作统计的，也就是说，要对它进行梯度上升的优化操作。用“白话”来说，就是把 θ 参数中的各个 w 的值挪来挪去，挪到一个扣着的锅盖的顶上去（读过本书前面内容的读者朋友应该知道我在说什么）。现在，我们只是不知道每次应该挪多少，也就是不知道 $\nabla J(\theta)$ 多大。因为知道了 $\nabla \text{Loss}(\theta)$ 多大，才能知道每次每个 w 各自要挪多少，所以，现在我们也需要知道 $\nabla J(\theta)$ 多大。怎么办？求导呗。

对 $J(\theta)$，需要进行改写，变成这个样子：

$$J(\theta) = E_{\tau\sim\pi_\theta}\left[\sum_t \tau\right] = \frac{1}{N}\sum_i \sum_t r\left(s_{i,t}, \pi(s_{i,t}|\theta)\right).$$

太好了，已经有 θ 出现了，也就是说，要想知道 $\nabla J(\theta)$ 多大，求 $\frac{\partial J(\theta)}{\partial \theta}$ 就可以了。

这种思路还是比较直接的，大致就是：把 θ 求出来，让机器人顺着一个以 s 和 a 构成的坐标系向前“爬”，纵坐标是奖励值 R。如图 9-1 所示（见彩插），一直在低处“爬行”的那条标有“×”的路径是不好的，而红色“山脊”一侧的位置比较好，那么就通过梯度上升的方式把路径优化到沿着“山脊”向上走的标有绿色“√”的路径上去吧。

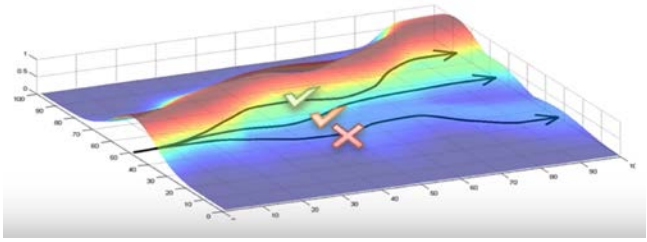


图 9-1　通过梯度上升求策略

9.2　DPG

2014 年，David Silver 发表了一篇名为 *Deterministic Policy Gradient Algorithms* 的论文。在这篇论文中，David Silver 证明了确定性策略梯度的存在，并把计算方法写了出来。

$$\begin{aligned}\nabla_\theta J(\mu_\theta) &= \int_s \rho^\mu(s)\nabla_\theta \mu_\theta(s)\nabla_a Q^\mu(s,a)|_{a=\mu_\theta(s)}\,\mathrm{d}s \\ &= E_{s\sim\rho^\mu}[\nabla_\theta \mu_\theta(s)\nabla_a Q^\mu(s,a)|_{a=\mu_\theta(s)}].\end{aligned}$$

这个公式是什么意思呢？就是策略梯度函数每次做优化的时候输出动作的网络 μ_θ 的 θ 的更新量有多大、怎么算。从本质上看，还是求偏导的问题。从形式上看，计算出来的 $J(\mu_\theta)$ 是一个动作的概率乘以动作估值的数学期望值，也就是 $\mu_\theta(s)$ 乘以 $Q^\mu(s,a)$。那么，求导呢？就是分别对它们的待定系数 μ_θ 中的 θ 去求导，后面的 $Q^\mu(s,a)$ 里面也嵌套了一个 $a=\mu_\theta(s)$。具体的计算和推导过程，将在 9.4 节介绍。

9.3 Actor−Critic

写到这里，一个重要的概念不得不提了，那就是 Actor-Critic。这个词组有时候会翻译成“演员-评论家”。这是一种有趣的模型进化方式，也可以说，这是一种模型训练的模式或技巧。

Actor-Critic 是 2000 年在 NIPS 上发表的一篇名为 *Actor-Critic Algorithms* 的论文中提出的。这篇论文论述的内容很多，核心就是提出了 Actor-Critic 模式（它不能称作“算法”，只能称作“模式”）。

在 Actor-Critic 模式的驱动下，模型的训练就分成了两个环节（或者说角色）。

一个是 Actor 角色，在一些资料中也称其为“演员角色”。这个角色是一个相对独立的模型。你仍然可以把它理解为一个神经网络，任务就是学动作。优化它的过程和优化一个普通 DQN 里面的网络没有太大的区别。

另一个是 Critic 角色，或者称作“评论家角色”。这个角色也是一个独立的模型。它的目的是学习估值的大小。在这种思维的指导下，估值学习也是一个独立的、可优化的任务，需要通过一个模型进行拟合。动作输出也是一个模型，通过一个模型进行拟合。

Actor-Critic 模式的优点和缺点都是比较明显的。

其优点是，更新是连续的（主要是指 Critic 角色），一般是通过梯度单步更新的，因此，更新的反馈比蒙特卡罗法要好（一个 Episode 结束才能进行更新）。此外，由于更新的连续性，方差也比较小，这样模型收敛的稳定性会高一些。

其缺点是，需要更新两个模型，且其中一个会直接影响另一个。想想看：如果 Critic 模型学出来的东西都是错的，那么 Actor 模型根本不可能学到一个好的动作策略。

Actor-Critic 模式虽然历史悠久，也要等到更厉害的“银弹”出现才能发挥作用。在 DDPG 模型中，就针对 Actor-Critic 的优点和缺点，做了不少 Trick，进行了不少优化。

9.4 DDPG

当我们理解了 DP 和 DPG 的概念之后，就不会觉得 Actor-Critic 模式有什么神秘的了。DDPG 就是 DPG 的深度学习网络模型（Deep Deterministic Policy Gradient）。这个模型的结构也不复杂，采用的是 Actor-Critic 模式（称为“A-C 模式”）。

说起来，A-C 模式并非深度强化学习独有的。那么，A-C 模式究竟是什么样的，又能带来哪些好处呢？单纯的文字叙述未免显得单薄，我们来具体了解一下 DDPG 模型的构造。

DDPG 的伪代码，如图 9-2 所示。又是一段很短的伪代码，但足以令人晕眩。说实在的，这些伪代码的抽象和复杂程度一点也不比真实代码低。

我试着把这些伪代码画成图，如图 9-3 所示，分成了两套模型、四个网络。

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q)^2)$
    Update the actor policy using the sampled gradient:

$$\nabla_{\theta^\mu}\mu|_{s_i} \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

图 9-2　DDPG 的伪代码

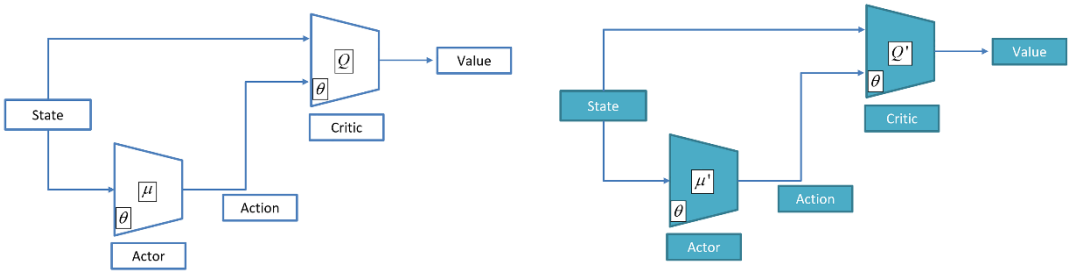


图 9-3　DDPG 模型示意图

左侧这套模型是在训练中“干活”（输出动作）的 Main Network（主网络）。右侧这套模型是 Target Network，即目标网络。我们通过阅读伪代码，看看在训练过程中都发生了什么。

四个小网络都初始化完毕。左侧这套模型中的网络先进行随机初始化。在这里，只有两个网络，也就是 $Q(s,a|\theta^Q)$ 和 $\mu(s|\theta^\mu)$。这种写法看上去确实让人感觉很严谨，不过对程序员来说，过多的上标和下标简直就是噩梦——明明是很简单的东西，非要描述得这么深奥。因此，我在图 9-3 中使用了简化的表示：网络 Q，输入是一个状态 s 和一个动作 a；网络 μ（也就是策略 π），输入是一个状态 s，输出是一个动作 a。这就是左侧模型中的两个网络的内容了。

μ 网络的工作方式看上去和 DQN 有几分相似——输入为状态 s，输出为动作 a。不一样的地方是，DQN 输出的动作是整个动作空间中每个离散动作各自的估值，而 μ 网络的输出是一个关于可选动作空间的动作概率分布或者一个连续的动作。总之，要做策略梯度，就必须使策略在梯度层面可优化。后面跟着的网络 Q，其输入由两部分组成，一部分是状态 s，另一部分是前面的 μ 网络输出的动作 a。

先不说别的，通过左侧模型中的两个网络的工作模式，你明白了什么呢？μ 网络根据输入的

状态 s 来拟合一个动作。Q 网络根据在状态 s 下做一个动作 a 来给出一个估值。就这么简单吗？如果感觉这个过程因为包含两个网络变量而变得复杂了，可以先“按住”一个不动，只看另一个。

假如前面的 μ 网络是准确的（即已经“学”到足够好的一个网络），那么，只需要训练后面的 Q 网络，而这就是一个回归问题了。也就是说，只评估在一个状态 s 下做一个动作 a 的估值，把 a 估算准就可以了。因此，MSE Loss 足够解决这个问题。后面这个估值的标签值是多少呢？可以用 $r_1+\gamma r_2+\gamma^2 r_3+\cdots$ 来表示。因为刚刚我们说得很清楚了，该表达式表示在某个状态的某个动作下获得的回报，所以通常也写成

$$J(\theta^\mu)=E_{\theta^\mu}[r_1+\gamma r_2+\gamma^2 r_3+\ldots],$$

也就是说，这个估值的大小应该是在以 θ^μ 为参数的 μ 网络在这个环境中不断“折腾”得到的样本的数学期望值。这个求数学期望值的过程，简单地说，就是求平均值，每一次的估值就用 $r_1+\gamma r_2+\gamma^2 r_3+\cdots$ 来表示（多次测量取平均值就能得到了）。所以，讲到这里你就会明白，Q 网络是一个单纯用于解决回归问题的网络。

再“按住”Q 网络不动，即认为它的估值已经很准了，不需要再训练了，让它提供准确的估值给前面的网络 μ，训练网络 μ 的动作输出能力。我们知道，网络 μ 输入的动作 a 的估值越高越好，最好每次输入的 a 都高过输出最高的估值。要知道，我们假设网络 Q 提供的估值是准确的，那么 a 作为 Q 网络的变量，是可以有确定的更新方向的。用术语表述，就是“a 的梯度方向是确定的”，或者“通过优化动作 a 满足在某个状态 s 下输出的估值最大”。动作 a 又是 s 的函数，也就是 $a=\mu(s|\theta^\mu)$。所以，在这个时候，所谓“优化 a”，就是对 θ^μ 进行梯度更新，让 θ^μ 向着满足 a 的变化趋势的需求进行更新，也就是说，如果网络 Q 的估值处在逐步估准的过程中，在一个状态 s 下就应该是一个有确定值的动作 a 或者概率输出，也就是说，如果在状态 s 下有一个动作的输出一定是最好的，那就提高输出它的概率。思路大致就是这样。

如果将两个网络“串联”起来，那么它们各自更新的梯度就是它们各自的待定变量的偏导数值了。对于求偏导数，我们也是“一回生，二回熟”，这一次就不觉得新鲜了。

首先，我们有估值的大小：

$$J(\theta^\mu)=E_{\theta^\mu}[r_1+\gamma r_2+\gamma^2 r_3+\ldots].$$

其次，我们有对前面的网络 μ 求导的链式法则：

$$\frac{\partial J(\theta^\mu)}{\partial \theta^\mu}=E_s\left[\frac{\partial Q(s,a|\theta^Q)}{\partial a}\frac{\partial \pi(s|\theta^\mu)}{\partial \theta^\mu}\right].$$

这样，我们就可以知道每次更新参数的量为多少了。

到这里，我们基本上把左侧模型中的网络（也就是主网络）的工作形式弄明白了，再回头看伪代码，就比较容易理解了。为了使结构看起来清楚一些，我们分三层来看。

（1）第一层

在第一层初始化了 $Q(s,a|\theta^Q)$ 和 $\mu(s|\theta^\mu)$ 两个网络（也就是主网络），并复制了和它们一模一样的两个网络 $Q'(s,a|\theta^{Q'})$ 和 $\mu'(s|\theta^{\mu'})$（也就是目标网络）。然后，初始化一个 Replay Buffer。

开启一个 *M* 次的循环，每次循环都启动一个 Episode，让机器人在里面玩游戏——玩到通关或者游戏结束为止。

（2）第二层

在第二层中，对每一个 Episode，在环境中一个机器人启动后的生命周期中，要做如下事情。

- 拿到这个 Episode 中的第一个状态 s_1。
- 启动一个 *T* 次的循环，每次迭代都是一个机器人的 Step。这个循环就是第三层。

（3）第三层

在第三层中，要对每个 Step 做如下操作。

首先，$a_t=\mu(s_t|\theta^\mu)+N_t$，就是用主网络中的网络 μ 输出一个动作，同时加上一个大小为 N_t 的值。在一些阅读材料中会把 N_t 解释为一种噪声，在伪代码中也称它为“noise”。别把它往坏处想，有噪声并不代表质量不好。使用它的目的，就是扩大取值范围，让 a_t 的输出试探更加丰富。

接着，把通过试探得到的 (s_t,a_t,r_t,s_{t+1}) 放到 Replay Buffer 中。需要注意的是，这时这个 Step 的主网络中的网络 μ 的任务已经完成了，它不会再做任何事情了。接下来的工作，就是进行更新计算。

有三个公式要放在一起看：

$$y_i=r_i+\gamma Q'(s_{i+1},\mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'});$$

$$L=\frac{1}{N}\sum_i(y_i-Q(s_i,a_i|\theta^Q))^2;$$

$$\nabla_{\theta^\mu}J\approx\frac{1}{N}\sum_i\nabla_a Q(s,a|\theta^Q)|_{s=s_i,a=\mu(s_i)}\nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}.$$

在第一个公式中，y_i 是一个目标值，从形式上看，就是 $r+\gamma Q(s,a)$，表示通过时间差分法来迭代更新，把当前的估值算准的过程。后面的部分看着有点乱，全是带“′”的项。这相当于，让如图 9-3 所示右侧模型中的网络 μ'，“假装”看到了这个状态，“假装”做这样一个动作，再由目标网络中的网络 Q' 对它进行评估，并把这个值作为目标值。毫无疑问，在一开始，这个值会非常不准，而随着试探的深入，这个值会逐渐变得准确起来。

在第二个公式中，L 表示主网络中的网络 Q 的损失函数，也就是 MSE Loss，目的是让它拟合自己和这个估值之间的距离。这个估值就是通过第一个公式计算出来的 y_i。前面的 $\frac{1}{N}$ 表示，这一次更新从 Replay Buffer 中挑选出来的样本数量 N，针对这个数量取平均值。

第三个公式可能是最长的更新公式了。$\nabla_{\theta^\mu} J$ 表示对主网络的 θ^μ 进行更新时的更新量，也就是 $\theta^\mu = \theta^\mu - \nabla_{\theta^\mu} J$。这个是整个 DDPG 模型的核心优化过程。至于后面的$\sum$部分，外面的一层是求平均值，我们主要看里面的一层。里面的一层，简单地说是一个链式法则，也就是

$$\frac{\partial J(\theta^\mu)}{\partial \theta^\mu} = E_s[\frac{\partial Q(s,a|\theta^Q)}{\partial a}\frac{\partial \mu(s|\theta^\mu)}{\partial \theta^\mu}]$$

在本模型中的写法。

在一个 Step 的最后，还要做一件重要的事情，就是完成这样的更新：

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'};$$

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}.$$

这里的 τ 是一个远远小于 1 的数字，在一些实验中会将其设置为 0.001。如果 τ 为 0.001，这个公式就会变得比较具象。在本例中，就变成了 $\theta^{Q\prime} \leftarrow 0.001\theta^Q + 0.999\theta^{Q\prime}$，也就是说，更新 $\theta^{Q\prime}$，但不是一下子粗鲁地把它赋值为 θ^Q，而是让它在很大程度上保留 $\theta^{Q\prime}$ 的值，然后慢慢地向 θ^Q 的方向更新。

如此这般，通过三个公式、两个循环，主网络就能用学会的一些行为做动作，在环境中逐渐试探，收集 Transition 了。在训练过程中，目标网络随主网络变化，同时为主网络提供估值的参考标准。损失函数都在主网络这一侧，主网络不断进行梯度更新并努力进化，慢性子的目标网络会跟随并学习。最终，目标网络会进化成我们需要的那个睿智的机器人。

老规矩，代码就不放在书中了，读者可以访问本书的 GitHub 页面下载。

9.5 本章小结

在这一章中，我们从 PG 讲到 DPG，最终讲到主角 DDPG。可以说，DDPG 是在单进程、单线程领域针对输出为连续维度的机器人的比较成熟的训练方案了，而且其思路不算复杂。

DDPG 的应用领域比较广，它的 A-C 模式的训练思想，也适用于很多复杂的游戏场景。本章通过一个简单的例子让大家明白：梯度上升的优化过程，可以在帮助一个 A-C 模式的训练模型逐步算准状态估值的同时，提高那些可以获得高回报估值的动作的输出概率。

第 10 章　A3C

在这一章中，我们来学习一下深度强化学习的第三类比较著名的模型，即 A3C（Asynchronous Advantage Actor-Critic，异步的优势演员-评论家算法）模型。

A-C 模式我们在第 9 章中介绍过了（DDPG 就是典型的 A-C 模式），相信大家已经有了比较完整的理解。本章将要介绍的“异步的优势演员-评论家算法”里的新鲜词是“异步”。

10.1　模型结构

A3C 的理念比较有趣。它希望通过多个进程或者线程的 Agent，在不同策略的指导下，不断在环境中试探，然后收集动作，最终“博采众长”，如图 10-1 所示。

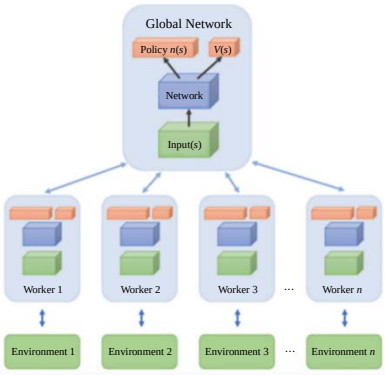


图 10-1　A3C 模型示意图

这种朴素的思维方式，似乎根本无须指定其蓝本是 DQN 还是 DDPG（两者都可以通过 A3C 得到效率上的提升）。没错，确实是这样的（虽然论文里不是这么称呼的）。

接下来，让我们一起通过 *Asynchronous Methods for Deep Reinforcement Learning* 这篇论文了解一下 A3C 版 DQN（A3C Q-Learning）和 A3C 版 DDPG（A3C Actor-Critic）是怎么工作的。

10.1.1　A3C Q–Learning

1. 伪代码

DQN 和 Q-Learning 我们已经很熟悉了。直接看伪代码，如图 10-2 所示。根据标题，这个算法的官方名称是“Asynchronous One-Step Q-Learning”（异步单步 Q-Learning）。这段伪代码是针对

每个 Agent 的，在这里称作 Actor-Learner 线程——叫什么不重要，看内容吧。

Algorithm 1 Asynchronous one-step Q-learning - pseudocode for each actor-learner thread.

// Assume global shared θ, θ^-, and counter $T = 0$.
Initialize thread step counter $t \leftarrow 0$
Initialize target network weights $\theta^- \leftarrow \theta$
Initialize network gradients $\mathrm{d}\theta \leftarrow 0$
Get initial state s
repeat
  Take action a with ϵ-greedy policy based on $Q(s, a; \theta)$
  Receive new state s' and reward r
  $y = \begin{cases} r & \text{for terminal } s' \\ r + \gamma \max_{a'} Q(s', a'; \theta^-) & \text{for non-terminal } s' \end{cases}$
  Accumulate gradients wrt θ: $\mathrm{d}\theta \leftarrow \mathrm{d}\theta + \frac{\partial (y - Q(s,a;\theta))^2}{\partial \theta}$
  $s = s'$
  $T \leftarrow T + 1$ and $t \leftarrow t + 1$
  if $T \mod I_{\text{target}} == 0$ **then**
    Update the target network $\theta^- \leftarrow \theta$
  end if
  if $t \mod I_{\text{AsyncUpdate}} == 0$ or s is terminal **then**
    Perform asynchronous update of θ using $\mathrm{d}\theta$.
    Clear gradients $\mathrm{d}\theta \leftarrow 0$.
  end if
until $T > T_{\max}$

图 10-2　A3C Q-Learning 的伪代码

首先，假设在全局范围内，主网络 θ 和目标网络 θ^- 是共享的，此外有一个全局性的计步器 $T = 0$。然后，初始化一个线程内的私有计数器 t，使 $t = 0$。将目标网络 θ^- 赋值为主网络 θ 的值。将更新主网络 θ 的梯度 $\mathrm{d}\theta$ 赋值为 0。

启动一个循环。这个循环一共要走 $T_{\max}$ 次。需要注意的是，这个循环的计步器 T 是全局共享的，也就是说，在训练整个模型的过程中可能启动了 5 个或者 10 个 Agent，但它们各自每一刻的 Step 都将计入 T。任何一个 Agent 在发现全局累积的步数大于最大的步数时就应该终止。

循环内部就是针对每一个 Agent 的内容了，我们来看看都做了些什么。

在循环内部，以概率 ε 做一个随机动作，以概率 $1 - \varepsilon$ 做从 $Q(s, a|\theta)$ 里查表得出的估值最大的动作。此时，会得到一个转移的状态 s' 和一个瞬时奖励值 r。

使用如下表达式计算目标值：

$$y = \begin{cases} r_j \\ r_j + \gamma \max\limits_{a'} Q(s', a'; \theta^-) \end{cases}.$$

对于终止的状态，使用瞬时奖励值就可以了。对于非终止的状态，要使用目标网络进行估算。

到目前为止，除了前面提到的全局共享主网络 θ 和目标网络 θ^- 的参数，其他部分与 2015 版 DQN 几乎没有区别。别急，大的区别还是有的，就是这个更新：

$$\mathrm{d}\theta \leftarrow \mathrm{d}\theta + \frac{\partial (y - Q(s, a; \theta^-))^2}{\partial \theta}.$$

在某个 Agent 的一个 Step 中，并没有像在 2015 版 DQN 中那样急着更新和优化主网络，即

$$\mathrm{Loss} = (y_i - Q(\phi_j, a; \theta))^2,$$

而是一步步累积需要更新的梯度 $\mathrm{d}\theta$。

周期性地对网络进行更新。当全局计步器 T 的值为 I_{target} 的整数倍时，将目标网络 θ^{-} 赋值为主网络 θ 的值。当本地计步器 t 的值为 $I_{\mathrm{AsyncUpdate}}$ 的整数倍或者当前 Agent 正处在一个 Episode 的结束时，用 $\theta = \theta - \mathrm{d}\theta$ 对主网络 θ 进行更新。

2. 特性

看完 A3C 版 DQN 的伪代码，不知道大家有没有发现其中有趣的地方。我自己有一些理解，与大家分享一下。

第一，A3C 版 DQN 的流程与 2015 版 DQN 非常接近。对于相似的地方，我们就不赘述了，主要看不同点。

第二，A3C 版 DQN 没有使用像 Replay Memory 这样的东西。在第 8 章中介绍过，2013 版 DQN 和 2015 版 DQN 都使用了 Replay Memory，主要目的是消除样本之间的关联性。样本之间的关联性对于估值计算来说是有害的，因为如果使用同一个策略或者从同一个 Episode 中走出来，序列 $s \to a \to r \to s'$ 之间的因果关系（或者说依赖性）会很强。而使用 Replay Memory 的作用就是将收集到的样本混在一起，随机抽取，进行贝尔曼方程的更新，从而有效地破坏相关性，使估值更为准确。在论文中，专门论述了不使用 Replay Memory 的原因，具体如下。

“Deep RL algorithms based on experience replay have achieved unprecedented success in challenging domains such as Atari 2600. However, experience replay has several drawbacks: it uses more memory and computation per real interaction; and it requires off-policy learning algorithms that can update from data generated by an older policy.

“In this paper we provide a very different paradigm for deep reinforcement learning. Instead of experience replay, we asynchronously execute multiple agents in parallel, on multiple instances of the environment. This parallelism also decorrelates the agents’ data into a more stationary process, since at any given time-step the parallel agents will be experiencing a variety of different states. This simple idea enables a much larger spectrum of fundamental on-policy RL algorithms, such as Sarsa, n-step methods, and actor-critic methods, as well as off-policy RL algorithms such as Q-learning, to be applied robustly and effectively using deep neural networks.”

以上内容的大意如下。

“基于 Experience Replay（就是指 Replay Memory），深度强化学习已然在很多领域的挑战中取得了空前的成功，例如 Atari 2600 游戏。然而，Experience Replay 有几个缺点：它占用了更多的内存空间，并且要在实时交互中使用更多的计算资源；它需要使用 Off-Policy 学习算法，这些算法可以从一个旧的策略生成的数据中更新而来。

“在本文中，我们提供了一种与以往有很大不同的深度强化学习模式，我们不用 Experience Replay 的方法，而在多个环境实例中异步并行执行多个 Agent。这种并行同样可以消除 Agent 的

数据相关性，并将其转化为更具稳定性的过程，因为在任何一个时间片上并行运行的多个 Agent 都将经历多样性极强的状态。这个简单的想法使得一个更大范围内的基础 On-Policy 强化学习算法成为可能，例如 SARSA 算法、n-Step 方法和 A-C 方法，当然也包括 Off-Policy 的强化学习方法，例如 Q-Learning。为了提高鲁棒性和效率，我们使用了深度神经网络。”

通过这些文字，我们了解了作者的观点与用意。应该说，A3C 是一种高效的方式。

第三，A3C 允许并行化。A3C 是第一个以显式方式表示可以使用并行化的强化学习算法，这绝对是值得肯定的。要知道，在线学习可以成倍提高样本收集效率，拿 CPU 换时间是非常划算的事情。这种并行化还可以“K.O.”Replay Memory，这虽然节省了存储空间，但会消耗一定的 CPU 资源。A3C 的并行化要收集 $s \to a \to r \to s'$，在这个过程中 GPU 是帮不上忙的，GPU 能帮忙的地方仅仅在

$$\mathrm{d}\theta \leftarrow \mathrm{d}\theta + \frac{\partial (y - Q(s, a; \theta^{-}))^2}{\partial \theta}$$

和

$$\theta = \theta - \mathrm{d}\theta$$

中。所以，要想让 A3C 的执行速度快一些，CPU 的处理速度和线程数量是关键条件。

第四，在单个 Agent 中对 $\theta = \theta - \mathrm{d}\theta$ 的更新是异步的，在全局范围内对 $\theta^{-} = \theta$ 的更新也是异步的。前面的学习经验告诉我们，异步的间隔越大，就越像离线学习，估值越稳定，收敛越平稳。所以，A3C 果断地采用了双重异步的方式，意在使网络的收敛更加平稳，而且在训练中可以根据网络的实际表现调整异步的节奏，例如 I_{target} 和 $I_{\mathrm{AsyncUpdate}}$ 的大小。这些值的意义，就是在 Agent 所表现出来的进步效率和收敛的稳定性之间找平衡，因此，我建议把这两个值设置得大一些。

A3C 算法的大致流程与原理，我们就了解这么多。读者可以访问本书的 GitHub 页面下载相关代码。

10.1.2 A3C Actor–Critic

除了 A3C One-Step Q-Learning 的伪代码，论文中还提到了 n-Step Q-Learning 的伪代码，在本书中就不详细介绍了。我们直接来看论文中给出的第三个伪代码段——A3C 版 DDPG 的伪代码，如图 10-3 所示。

这段伪代码的蓝本是 DDPG。如果大家还有印象的话，我们在 9.4 节中介绍过，DDPG 是一个比较“特殊”的网络——不“训”则已，一“训”就是四个（应该说是两组网络，一组是主网络，另一组是目标网络）。主网络里有一个 Actor 和一个 Critic，目标网络里也有一个 Actor 和一个 Critic。那么，在 A3C 版 DDPG 中，形式也是一样的。为什么形式会是一样的呢？因为目标网络是共享的。

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ and θ_v and global shared counter $T = 0$
// Assume thread-specific parameter vectors θ' and θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
　Reset gradients: $\mathrm{d}\theta \leftarrow 0$ and $\mathrm{d}\theta_v \leftarrow 0$.
　Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
　$t_{\text{start}} = t$
　Get state s_t
　repeat
　　Perform a_t according to policy $\pi(a_t|s_t;\theta')$
　　Receive reward r_t and new state s_{t+1}
　　$t \leftarrow t+1$
　　$T \leftarrow T+1$
　until terminal s_t **or** $t - t_{\text{start}} == t_{\max}$
　$R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t,\theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
　for $i \in \{t-1,\ldots,t_{\text{start}}\}$ **do**
　　$R \leftarrow r_i + \gamma R$
　　Accumulate gradients wrt θ': $\mathrm{d}\theta \leftarrow \mathrm{d}\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
　　Accumulate gradients wrt θ'_v: $\mathrm{d}\theta_v \leftarrow \mathrm{d}\theta_v + \partial\,(R - V(s_i;\theta'_v))^2/\partial\theta'_v$
　end for
　Perform asynchronous update of θ using $\mathrm{d}\theta$ and of θ_v using $\mathrm{d}\theta_v$.
until $T > T_{\max}$

图 10-3　A3C 版 DDPG 的伪代码

首先，设置一个全局共享的目标网络，Actor 的参数为 θ，Critic 的参数为 θ_v，下标 v 的意思是“Value”（表示用它来做估值）。每个 Agent 的 Actor 参数为 θ'，对应的 Agent 的 Critic 网络的参数是 θ'_v。这个结构是对称的，很容易记忆。然后，仍旧是全局计步器和本地线程计步器（分别初始化为 $T=0$ 和 $t=1$）。开启一个循环，直到 $T > T_{\max}$。

循环内部的工作流程也比较简单。

首先，把全局目标网络中 Actor 的参数 θ 的更新量 $\mathrm{d}\theta$ 设置为 0，把 Critic 的参数 θ_v 的更新量 $\mathrm{d}\theta_v$ 设置为 0。同时，把线程本地的 Actor 的参数 θ' 赋值为目标网络的 Actor 的参数 θ 的值，把线程本地的 Critic 的参数 θ'_v 赋值为目标网络的 Critic 的参数 θ_v 的值。

这里有一个循环，就是在当前线程中让机器人不停地玩游戏，直到通关、游戏结束或者达到试探步数的上限（$t - t_{\text{start}} = t_{\max}$）为止。在它“生存”的过程中，使用策略 $\pi(a_t|s_t;\theta')$ 输出动作，也就是用本地的 Actor 网络来输出动作。同时，要记录相关的 $s \to a \to r \to s'$。

以第二个循环结束时遇到的状态 s_t 为研究对象，先计算估值。使用表达式

$$R = \begin{cases} 0 \\ V(s_t,\theta'_v) \end{cases},$$

0 表示 s_t 为终止状态 Terminal，$V(s_t,\theta'_v)$ 表示其他状态。这一点，不仅和 DQN 不一样，也和 DDPG 不一样。DQN 在很大程度上还要指望在处于 Terminal 状态时所赋予的一个很大或者很小的奖励值，以训练机器人找到方向，但是在这里，只要游戏结束，得分都是 0。

在这里，学习与更新也构成了一个循环，循环范围是 $t-1$ 到 t_{start}。可以看出：t 这个计步器的值是在当前线程中不断增加的，而且从不清零；t_{start} 是在机器人玩游戏之前记录的步数，所以当机器人开始玩游戏之后，$t-1$ 的值肯定要比 t_{start} 的值大一些。这个循环可以认为是一个从大到小的倒序循环，即先处理距现在最近的那个时间片。这样处理的目的，主要是计算更为方便，

因为根据蒙特卡罗法，某个时间片 t 的状态估值是逐层“打折扣”的：

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^{T-1} r_T.$$

如果从后面开始算，就可以不使用递归的方式来表示，而直接使用

$$R = r_i + \gamma R$$

来表示。经过 $t-1$ 到 t_{start} 的循环过程，自然就能得到上面那个 G_t 的表达式。

接着是梯度更新：

$$\mathrm{d}\theta \leftarrow \mathrm{d}\theta + \nabla_{\theta'} \log \pi\left(a_i|s_i;\theta'\right)\left(R - V(s_i;{\theta'}_v)\right);$$

$$\mathrm{d}\theta_v \leftarrow \mathrm{d}\theta_v + \frac{\partial (R - V(s_i;{\theta'}_v))^2}{\partial {\theta'}_v}.$$

这两个公式看起来有点吓人，不过仔细看看——是“老熟人”。

这两个公式表示对目标网络中的 Actor 的参数 θ 的更新大小，是一个累积的量，从 $t-1$ 到 t_{start} 的每个时间片都要经过这样的累积。每次累积的量是本地线程使用的 Actor 的 θ' 梯度，这个梯度的大小是

$$\pi(a_i|s_i;\theta')\left(R - V(s_i;{\theta'}_v)\right).$$

然后，在外面取对数（这部分可能不太容易理解）。$\pi(a_i|s_i;\theta')$ 是一个概率值，$(R - V(s_i;{\theta'}_v))$ 是估值的误差。

再看一下 Critic 网络的更新，只是一个大小为 $\frac{\partial (R - V(s_i;{\theta'}_v))^2}{\partial {\theta'}_v}$ 的梯度。也就是说，θ 的更新量 $\mathrm{d}\theta$ 是沿着策略 $\pi(a_i|s_i;\theta')$ 的参数 θ' 的变化方向变化的，${\theta}_v$ 的更新量 $\mathrm{d}{\theta}_v$ 是沿着减小估值误差的 ${\theta'}_v$ 的变化方向变化的。因为 $\pi(a_i|s_i;\theta')$ 的取值范围仅为 0 到 1，直接求梯度会使其消失，所以在这里也算使用了一个 Trick，即在求对数之后，原来在 0 到 1 之间的分布直接变成了在正无穷到 0 之间的分布，梯度（偏导数）一下子变大了。由于有梯度就会有更新，网络学东西的速度也会快一些。

在循环的最后，会进行一次更新。全局性的 Actor 网络和 Critic 网络的更新，相当于只要机器人“玩”过一个 Episode，就让全局网络对它所学到的东西进行一次学习，且更新是以 Episode 为单位进行的。

在本书的 GitHub 页面上提供了一段很短的代码，结合本节的内容不难理解。

10.2 本章小结

当你认识了 A3C 版 DQN（A3C Q-Learning）和 A3C 版 DDPG（A3C Actor-Critic）之后，就可以考虑用这类方式代替原始的 DQN 和 DDPG 来进行 Gym 游戏的训练了。毫无疑问，从对空间进行试探的角度看，这种方式的效率很高，可以充分发挥多 CPU 计算机的性能（如图 10-4 所示）。

由于 Gym 中的游戏大部分都是单线程的，从理论上讲，对这种情况确实可以考虑多开几个线程，尽量把 CPU 用“满”。这相当于提高了样本收集和网络“正向传播”的效率。

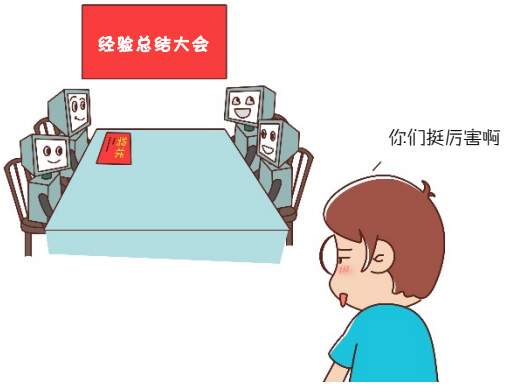


图 10-4　A3C 的并行能力强

Python 在多线程这个环节中存在一些问题——与 GIL 有关。GIL 的全称是“Global Interpreter Lock”，译为“全局解释器锁”。为了发挥多核 CPU 的性能，程序多采用多线程/多进程的模式。多线程模式必然涉及线程之间的通信或者同步问题，而最简单解决方法就是添加互斥锁（GIL 就是用来实现这一点的）。可是，GIL 在设计上存在缺陷，在多核的 CPU 上表现糟糕。而且，GIL 出现的时间很早，大量模块都使用了它，所以，后来当开发者发现 GIL 的效率存在严重问题也为时已晚。现在，GIL 算是 Python 的一个“历史遗留问题”，要想“根治”它，恐怕很难了。

如果想在 Python 3 中使用 A3C 进行训练，我建议还是采用“多进程+多协程”的方式。需要注意的是，训练时使用的游戏环境也要支持这种方式。如果不能启动多个副本，是很麻烦的，可能你不得不使用 Docker 或者 KVM 的虚拟容器来隔离它们，并要通过网络连接的方式让它们共享权值。这些都属于工程问题，虽然不是算法模型本身的应用技巧，但对于提高训练效率、充分利用硬件资源是大有裨益的。如果有精力，最好多了解一些。

第 11 章 UNREAL

在 2016 年，在 A3C 之后，又一个颇为有趣的模型出现了，那就是 UNREAL。

UNREAL 来自论文 *Reinforcement Learning with Unsupervised Auxiliary Tasks*。在这篇论文里，提出了一种叫作“UNsupervised REinforcement and Auxiliary Learning”（也就是所谓“无监督强化辅助学习”）的模型，“UNREAL”就是其中部分单词打头几个字母的缩写。幸亏是缩写——词组这么长，确实不太好记……

这篇论文的篇幅有 14 页 A4 纸，洋洋洒洒，论述颇为精彩。在本章中，就和大家分享一下这篇论文的核心思想。

UNREAL 模型分为四部分，如图 11-1 所示。

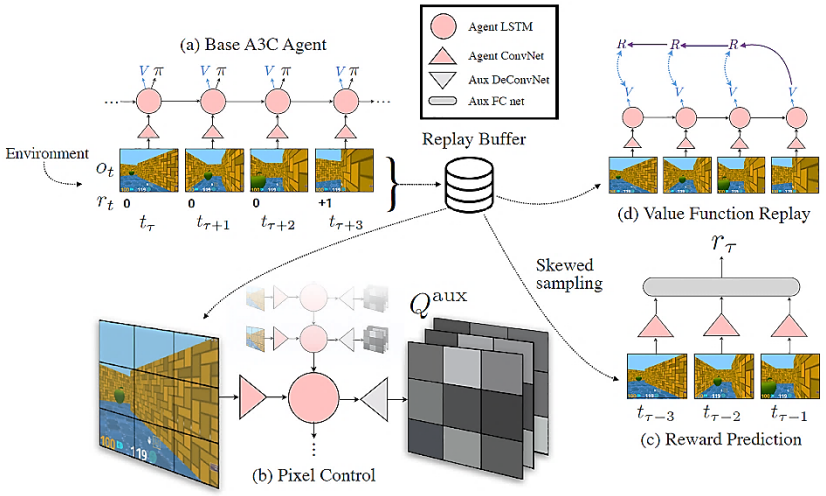


图 11-1 UNREAL 模型的结构

11.1 主任务

如图 11-1 (a) 所示，就是主任务部分。它是一个普通的基于 A3C 模型的 Agent，网络结构为 CNN-LSTM，也就是一个前半部分为卷积神经网络、后半部分为 LSTM 的网络结构。

主任务部分的训练方式和 A3C 模型的训练方式没有本质差别，使用的损失函数也是 A3C 模型的损失函数。唯一值得一提的可能就是 LSTM 部分了。根据我们掌握的深度学习的相关知识，它带来的好处就是能记忆这一帧的前 *n* 帧内容对当前内容的影响。对这种连续的过程中包含的信

息，还是要保留下来的。传统的 DQN 模型使用 4 个连续的帧组成了一个完整输入模型的 x，使用 LSTM 网络来实现记忆。

在这里插入一段“短评”——

现在越来越多的学者发现，LSTM 网络不仅训练难度非常大，还存在占用空间大、收敛速度慢、容易产生过拟合等一系列缺陷，在很多场景中让人非常头疼。所以，也有一种趋势是：能使用 CNN 解决的场景，尽量使用 CNN 解决，不要动不动就使用 LSTM。

我觉得这段评价还是有一定道理的，也感同身受，所以，放在这里供大家参考。在实际的工程中，还是优先使用 CNN 来解决问题。

回到正题。在图 11-1 (a) 部分中，长边在下的三角形代表卷积神经网络层，圆形代表 LSTM 网络层。在最后一层 Feature Map 的后面，有两个不同的后端网络：一个是 V 网络，用来拟合状态估值；另一个是 π 网络，用来学习策略。这部分的 4 幅图片为 t_τ 到 $t_{\tau+3}$，表示连续的 4 帧按照时间序列逐一输入网络。在这个过程中收集到的 Transition 信息，会被放到 Replay Buffer 中。

到这里为止，还没有和 A3C 模型差别明显的内容出现。

11.2　像素控制任务

如图 11-1 (b) 所示，就是像素控制任务（Pixel Control）。在这个部分，要训练一个辅助的策略网络 Q^{aux}，希望能够通过训练把输入图片的不同区域的像素强度（Pixel Intensity）最大化。从形式上看，这部分就是使用一个 Encode 和一个 Decode，对从 Replay Buffer 中取出的一帧图像进行处理的过程。Encoder 代表一个卷积神经网络层和一个 LSTM 网络层，后面的 Decoder 则代表一个反卷积（Deconvolutional Network）层。

在这里，有必要介绍一下反卷积的概念。反卷积也叫作转置卷积（Transposed Convolution），是一种升采样（Upsampling）手段，如图 11-2 所示。在 6.10 节中，我们讲解过使用 Max Pooling 和 Average Pooling 的手段做降采样、通过池化层来降维的方法。反卷积的目的与此相反，它是用来做升维的。如果你需要将一个低维空间中的矩阵或者向量映射到高维空间中，那么反卷积是一种经常会使用的手段。

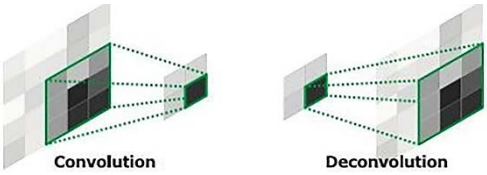


图 11-2　反卷积示意图

在我看来，“反卷积”这个名字似乎有点不科学，因为它并不是卷积的逆过程。我们来看一个例子，如图 11-3 所示。

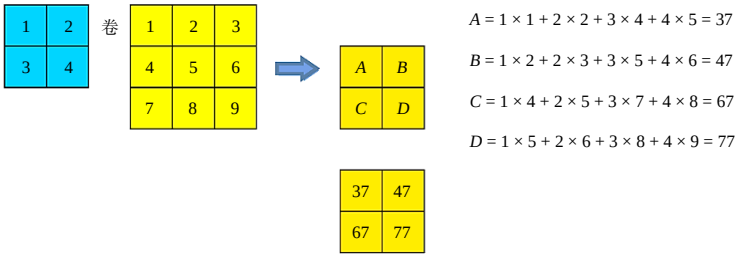


图 11-3 卷积计算

通过第一个权重分别为 [1,2,3,4] 的卷积核，对一幅图片进行卷积操作，形成后面的 Feature Map，那么，在这一层得到的值就是 [37,47,67,77]。这就是卷积过程。

如果你希望训练一个反卷积神经网络，把这个降到四维的向量还原到九维，可以吗？并非只有通过池化层做了降采样的向量才能做反卷积，反卷积是为了做升采样而使用的，是为了实现一种映射运算而构造的矩阵乘法。

通过这个例子，如果把卷积计算“展开”，变成矩阵乘法运算，应该是什么样的矩阵相乘呢？根据矩阵乘法运算的定义，展开后的运算，如图 11-4 所示。

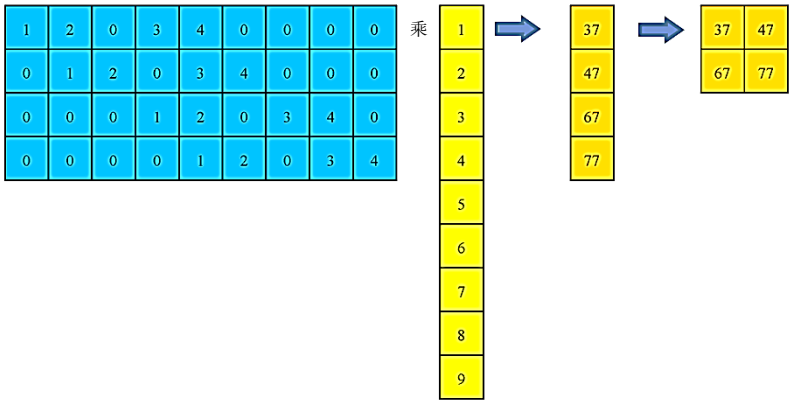


图 11-4 将卷积计算转换成矩阵乘法

大家看看，如图 11-4 所示的计算和如图 11-3 所示的计算是不是等价的？

如果我们把一个四维的向量还原到 3×3 的九维向量空间中，就需要对这个卷积核进行转置，把原来 4×9 的矩阵转换成 9×4 的矩阵。这时，再和一个 4×1 的矩阵相乘，就能得到一个 9×1 的矩阵。重组这个 9×1 的矩阵，就能得到 3×3 的矩阵了。这就是反卷积过程，如图 11-5 所示。

不过，在这里，对转置后的矩阵，我们只需要它的形状。也就是说，我们需要它的形状从 4×9 变成 9×4，这样才能构造出后面矩阵乘法的结果，即 9×1——这是关键。而将原来的卷积核转换成矩阵，然后进行转置，得到的新矩阵是不能直接通过运算得到结果的。

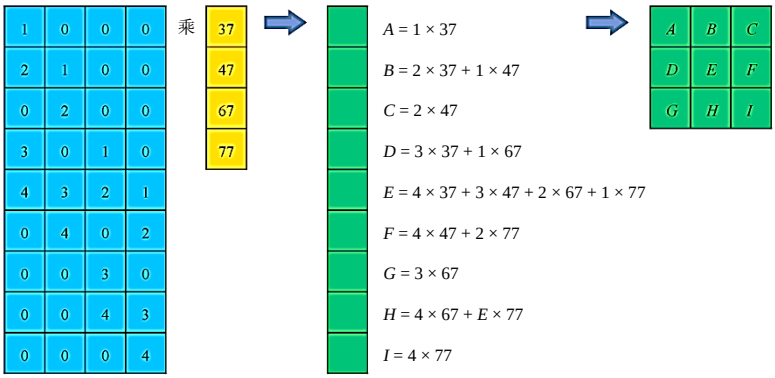


图 11-5　反卷积

就以如图 11-5 所示的过程为例，$A \sim I$ 的值是 1 ~ 9 吗？显然不是。因此，大家记住结论就可以了：在进行升采样操作时使用反卷积，反卷积的卷积核（也就是那个转置矩阵）的形状是可以使用的；但是，对于一个监督学习过程，期望的输出值是通过后面的升采样得到的矩阵，它的值是标签，需要通过训练得到的是这个转置后的矩阵，即通过梯度下降法这一类的优化算法得到一个使得升采样过程的误差尽可能小的转置矩阵。

在这个任务中，网络的前半部分其实和主任务共享一套 CNN 和 LSTM，只是后半部分的反卷积神经网络不一样。

后半部分最终输出一个 $N_{\text{act}} \times n \times n$ 的张量，称为 $\boldsymbol{Q}^{\text{aux}}$，$N_{\text{act}}$ 表示动作空间的数量（在迷宫游戏里，应该是前、后、左、右 4 个不同的动作）；$n \times n$ 表示像素块（也就是图 11-5 右侧那个 3×3 的划分方法）。三维矩阵 $\boldsymbol{Q}^{\text{aux}}$ 的含义是什么呢？三维矩阵 $\boldsymbol{Q}^{\text{aux}}(\boldsymbol{a}, i, j)$ 中的 $\boldsymbol{a}$、i、j 坐标点表示，在做了动作 $\boldsymbol{a}$ 之后，在 $\boldsymbol{a}$ 所表示的那个二维矩阵中，第 i 列第 j 行的像素块的变化究竟有多大。有了以后这个定义以后，$\boldsymbol{Q}^{\text{aux}}$ 的构造就不是什么难事了。对于 $N_{\text{act}} \times n \times n$ 中的每个像素块来说，都可以使用类似 MSE Loss 的方法求出两帧对应像素块之间的差。

论文中还提到，这个任务是为了让机器人学会控制环境而设计的。不过，这一点到目前为止还看不出来，我们需要继续往后看。但是，从总体上看，如果图像的变化比较大，就说明机器人正在经历比较重要（或者说重大）的选择。

对于论文中提到的那个让机器人在迷宫里面转悠的场景，如果图像的变化比较大，就不会沿着迷宫往前直走，十有八九会寻找其他出路。可以将这样的时刻理解成面临重大选择的时刻。

11.3　奖励值预测

如图 11-1 (c) 所示，叫作奖励值预测任务。根据邻近的 3 帧图像 $\tau-3$、$\tau-2$、$\tau-1$，预测 τ 这一帧图像的奖励值 Reward（当然，这时 τ 这一帧还没有出现）。这个任务也需要由一个网络来完成，因此，这是一个典型的回归问题。输入连续的 3 帧图像，给出一个确定的 Reward 值来进

行拟合。对于从本书开头一直读到此处的你来说，这真的不足为奇了——这就是一个典型的监督学习场景。

网络的前半部分也使用了主任务的 CNN，而后半部分不是共享的了，专门负责表示从 CNN 提取后的特征到 Reward 值的函数映射。论文中说，希望通过这样的方式来消除感知奖励值的稀疏性。而我个人的理解是：可以通过对前面几帧（或者说状态）的观测来预测即将到来的奖励值。这是一个对奖励预判能力的学习过程。既然是一个回归问题，对这个子任务来说，使用 MSE Loss 就足够了。

论文中还专门配了一幅图，如图 11-6 所示，左边表示机器人在迷宫中的视角，右边表示“上帝视角”。这幅图主要为了说明一种非常典型的情况：如果吃到前面的苹果，可以得到 +1 的奖励值；如果还能吃到更远处有点像糖葫芦的东西，可以得到 +10 的奖励值；在将后者吃掉后，游戏就会重新开始。

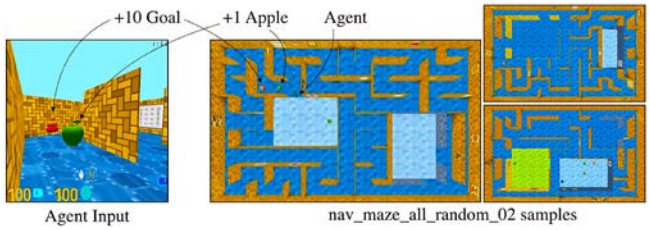


图 11-6　UNREAL 的奖励值预测

这个“小人物”如果训练成功了，机器人就会进行预判：如果往前走，很快就能获得 +1 和 +10 的奖励值——“棒棒哒”。因此，这时它就会倾向于往这个方向走。这就是这个子任务的用意。

11.4　值函数回放

如图 11-1 (d) 所示，叫作值函数回放（Value Function Replay）。论文里并没有给出实现值函数回放的具体方案，但是对几个要点的说明是比较清楚的。论文作者对收集到 Replay Memory 中的 Transition 信息做了分类。分成了两类：一类叫作“Rewarding Subset”，即带有回报的 Transition；另一类叫作“Non-Rewarding Subset”，即无回报的 Transition。要知道，游戏中并非苹果满地，所以，如果在一个“迷宫”里“走夜路”，获得 Reward 将是小概率事件。能够获得回报的样本如此稀疏，很难学到东西，因此，在这个 Case 中作者采用了过采样（Oversampling）的方式。

过采样是一种处理分布较偏的样本的技巧。

假设有这么一个场景：需要训练一个能够区分猫和狗的图片的神经网络，你的手里有 950 张猫的图片，50 张狗的图片。这就是样本分布较偏的情况，直接训练模型不会得到好结果。如果用交叉熵函数作为损失函数，会发生什么呢？每个样本正确或错误的权重是相等的。假如我们能够直接把神经网络初始化成一种“神经”的状态，例如，将所有的图片输入后，输出都是猫的分类，

那么，在这个包含 1000 张图片的集合中，可以得到一个准确率高达 95% 的模型——猫确实都判断对了，而可怜的“汪星人”一个也没有被识别出来。这是一个好模型吗？肯定不是，因为猫的特征根本就没有被学到（即使不学，也已经有 95% 的正确率了）。

因此，这种时候就要考虑使用过采样。最直接的方法就是把每一张狗的图片复制 18 份，这样原来的 50 张狗的图片就变成 950 张了。将这 950 张狗的图片和 950 张猫的图片混在一起进行训练，情况确实会有很大的改观，起码对于猫的特征提取来说，神经网络会显得“认真”“谨慎”很多。然而，虽然狗的图片的数量变多了，但单纯的数量增加并不能使样本集中蕴含的特征变得丰富，而且对于狗的图片，确实更容易出现过拟合的情况。因此，这种方法有很大的局限性。如果实在没有其他办法，这样处理会比直接拿偏样本来训练好一点（但是效果有限）。

为此，科学家们设计了一种叫作 SMOTE 的算法来进一步增强过采样的效果。“SMOTE”是一个缩写词，全称是“Synthetic Minority Oversampling Technique”，直译过来就是“综合少数样本过采样技术”。

大致介绍一下 SMOTE 算法的计算过程。一个类的样本属于少数类，有 T 个样本，如果需要把它变成 NT 个样本，就要从这 T 个样本中找一个样本 x_i，并在这 T 个样本中使用 kNN 算法找到离 x_i 最近的 k 个样本。从这 k 个样本中随机选择一个样本 x_j，再随机生成一个 0 ~ 1 之间的数字 a。此时，会生成一个新样本

$$x_1 = x_i + a(x_j - x_i),$$

也就是

$$x_1 = (1 - a)x_i + ax_j.$$

这个新样本 x_1 就相当于一个被“造”出来并“安插”在两个 *T* 类别样本之间的样本。这其实是插值法的思想，a 的大小决定了新样本离两者中的哪一个更近。如果能“造”出 1 个样本，那么再“造”出 100 个样本也不成问题。这种技巧，在其他机器学习场景中也会使用。

11.5　损失函数

在了解了各个辅助任务的目标之后，让我们来看一看损失函数。想必很多读者朋友已经猜到了，要训练这样一个多任务、多目的的网络，损失函数中要有多个评价项目，以分别满足这些辅助任务的需求。

$$L_{\mathrm{UNREAL}}(\theta) = L_{\mathrm{A3C}} + \lambda_{\mathrm{VR}} L_{\mathrm{VR}} + \lambda_{\mathrm{PC}} \sum L_Q^{(c)} + \lambda_{\mathrm{RP}} L_{\mathrm{RP}}$$

这个损失函数中的项目看上去很多，但是仔细看看，其实并不复杂。既然使用这样一个损失函数来训练网络，那么对我们来说，它的每一项的值越小越好。

（1）L_{A3C}

A3C 模型训练的损失部分，表达式为

$$L_{\mathrm{A3C}} \approx L_{\mathrm{VR}} + L_{\pi} - E_{s\sim\pi}[\alpha H\big(\pi(s,\cdot,\theta)\big)].$$

这个表达式的意思是，损失值 L_{A3C} 约等于右边这些约束条件的和。

约束条件都有哪些呢？估值损失、策略损失——其实就是由 Actor 和 Critic 不准导致的损失。最后一项 $-E_{s\sim\pi}[\alpha H(\pi(s,\cdot,\theta))]$ 指的是熵。从表达式的角度看，希望这一项尽可能大，这样得到的结果才会尽可能小。期望值大的部分是 $H(\pi(s,\cdot,\theta))$，即在其他条件都能得到保证的情况下，还是希望在输入状态 s 的时候，网络能够输出较为丰富的、不确定性较高的动作策略。

（2）L_{VR}

Replayed Value Loss 的表达式为

$$L_{\mathrm{VR}} = E_{s\sim\pi}[(R_{t:t+n} + \gamma^n V(s_{t+n+1},\theta^-) - V(s_t,\theta))^2].$$

简单地说，就是对状态估值准确性这个子任务的收敛约束。

（3）$\sum L_Q^{(c)}$

Q-Learning Loss 的表达式为

$$L_Q^{(c)} = E[(R_{t:t+n} + \gamma^n \max_{a'} Q^{(c)}\,(s',a',\theta^-) - Q^{(c)}(s,a,\theta))^2].$$

这就是贝尔曼方程的收敛表达式，即一个 n-Step 的 Q-Learning 做估值回溯的收敛约束。右上角的 (c) 是对某个辅助控制任务（Auxiliary Control Task）而言的。

（4）L_{RP}

论文中虽然没有给出 L_{RP} 的表达式，但说明了这个东西是“Auxiliary Reward Task”这个子任务的损失项。那么，我们认为它是一个纯粹的 MSE Loss 即可。标签值就是应该学到的 Reward 值（也就是预测值）。

看上去有点复杂，不过，根据我们训练深度学习网络的思路，只要能够确定输入、输出、网络结构、损失函数，就可以开始训练了。反正在网络中，除了必要的 A3C 模型训练的约束，人类会“强迫”机器人注意若干事项——让机器人注意那些应该注意的问题，从而提高收敛效果。

在写作本书时，代码调试尚未结束，所以请大家关注本书 GitHub 页面的更新。我们会在调试完成后的第一时间把代码上传。

11.6 本章小结

在本章中，我们一起了解了 UNREAL 算法模型的思路。

我不敢说 UNREAL 模型的有效性究竟有多高，但它确实足够有趣。UNREAL 模型期望通过

一些辅助性质的工作来尽可能实现自动化的高效训练过程。体现在损失函数上，就是损失函数有很多分项，而这相当于给一个训练过程引入了更多的限制条件。这种思路是非常值得我们借鉴并在未来其他模型的训练过程中大胆尝试的。

论文中还提供了一些对比数据，如图 11-7 所示。UNREAL 在 Labyrinth 任务中的表现不错；但在 Atari 游戏中，UNREAL 的表现与 A3C 相比，没有明显的提升。

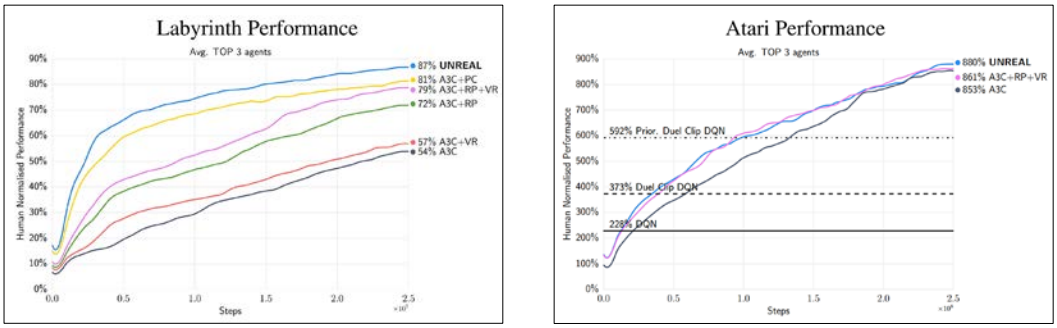
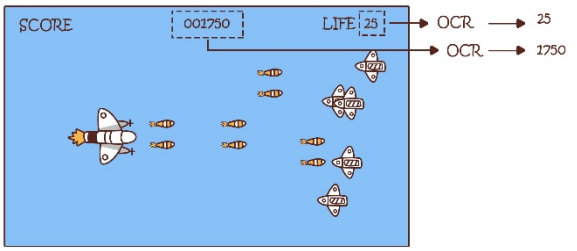


图 11-7　UNREAL 的得分能力

有关深度强化学习模型算法方面的介绍，到此就告一段落了。在“扩展篇”中，我们将讨论一些令人“脑洞”大开的话题。

扩展篇



第 12 章 NEAT

谈到人工智能，大家通常会不由自主地想到深度学习。确实，很多深度学习项目令我们耳目一新。研究得更为系统的朋友，会选择关注强化学习，因为这种解决贯序决策问题的方法，能够解决很多纯粹的基于深度学习的方法所不能解决的问题。此外，一种新的建模理念也得到了迅速发展，那就是 NEAT（Neuro Evolution of Augmenting Topologies，增强拓扑神经网络）。

NEAT 与普通深度学习模型解决问题的思路不太一样。

一般的深度学习模型解决问题的思路是，期望搭建一个深度神经网络来拟合一个从输入 x 到输出 y 之间的复杂的映射关系。通过深度神经网络中丰富的神经元的组合，以及一个个小的、拟合能力有限的功能组件，构合成一个拥有复杂拟合能力的模型。网络比较深是这类模型的共同特点。因为在一般情况下，越深的网络，其 VC 维越高，高层特征的提取能力越强，网络能够记忆的内容越多。

而 NEAT 的方法论与深度学习模型不同。与其说 NEAT 是强化学习的延伸，还真不如说它是神经网络和遗传算法的朴素结合。所以，严格地说，在强化学习的书籍中介绍 NEAT，显得有些奇怪。但是，NEAT 是一种新颖的且在一些环境中非常有效的方式，不提我又有些意犹未尽，而在这本书中“乱入”NEAT 也是有益无害的——至少对曾广思路是有好处的。

我们简单了解一下 NEAT 的建模思路。假设有一个以状态 s 作为输入、以动作 a 作为输出的神经网络 $a = f(s)$。这个神经网络能够把输入的状态 s，经过适当的网络拓扑结构中的层层连接，映射到输出的动作 a 上去。把这个连接的模型放到真实环境中进行测试，看看它的表现到底怎么样。对于那些表现好的拓扑基因，我们把它们留下或者以大概率选择它们；对于那些表现不太好的拓扑基因，我们将它们丢弃或者以小概率选择它们。让这些拓扑连接起来并进行“杂交”，产生“下一代”的连接方式。通过这样的迭代“进化”，优胜劣汰，神经网络总会大概率向着好的方向发展。

听起来很美妙吧？根据我的经验，NEAT 的优点有如下三个。

（1）拓扑连接变得简单

拓扑连接一旦变得简单，有一个优点大家自然能够想到，那就是存储这个模型需要的空间会减小。

（2）可解释性变强

当拓扑变得简单以后，模型的可解释性也会得到增强。回顾我们训练过的深度学习模型，通

常层数比较多、映射关系比较复杂，因此，很难从模型的结构上解释每一层究竟提取了什么、实现了哪些具体维度上的采纳和计算。NEAT 产生的网络连接简单，有限的连接带来了较强的可解释性，这一点毋庸置疑。

（3）在一些场景中收敛极为迅速

我们惊喜地发现，在一些训练环境中，NEAT 的收敛速度很快。虽然并非在每个场景中都如此，但这仍然让我们眼前一亮。

既然 NEAT 是由神经网络和遗传算法组合构成的，那么对这两个概念我们都需要有所了解。在“现代篇”中已经对神经网络进行了相对系统的介绍，下面我们就来认识一下遗传算法。

12.1　遗传算法

与其说遗传算法是一类算法，不如说是它一种处理问题的思想。遗传算法的整个体系都在表达对于某种问题的处理思路和原则，而不是一个具体的代码编写过程。

遗传算法（Genetic Algorithm）是一类借鉴生物界的进化规律（适者生存、优胜劣汰的遗传机制）演化而来的随机化搜索方法。它是由美国的 J.Holland 教授于 1975 年首先提出的。遗传算法借鉴的可是进化论的理论依据。在遗传算法的体系里，思维方式远比编写代码重要，所以，让我们先用一点篇幅来缅怀一下遗传算法“真正的鼻祖”，著名的英国生物学家查尔斯·罗伯特·达尔文——进化论的奠基人。

进化论不是一本书，而是一个有关物种发源与发展的逻辑体系。达尔文在 1859 年出版了《物种起源》，由此开创的“物竞天择，适者生存”的进化论体系被科学界公认为 19 世纪自然科学的三大发现之一（另外两大发现是细胞学说和能量守恒与转化定律）。达尔文病逝于 1882 年 4 月。为了表示对这位科学家的崇敬之情，人们把他的墓放在牛顿墓旁边——位于英国伦敦的威斯敏斯特大教堂（也就是我们平时所说的西敏寺）。

12.1.1　进化过程

言归正传，我们说说这类算法的思路是怎样的。在物种进化过程中，一开始，地球上只有海洋，没有陆地，从单细胞动物到鱼，不知道进化了多少代。陆地出现以后，第一批想上岸去“谋生”的鱼（谋生的具体动机可能不太明确）由于种种原因，和其他的鱼有了差别，例如“胆儿肥”（最起码敢上岸去玩耍）、有个能直接呼吸空气的鳃（能在空气中存活）、有格外强壮的鳍（能“跑”得快一些）。后来，这些“上岸”的鱼中，有一些就有了陆地生活格外倚重的器官，例如粗壮的鳍、空气摄入能力更强的肺、视力更好的眼睛等。如果这些器官的功能变得强大了，就会为它们在陆地上的生活提供更好的支持。这算也是“生活所迫”吧——要么生存，要么死亡，要么回家。只有具有适应能力的种群，才能在相应的环境里生存。

根据进化论的观点，“物竞天择，适者生存”，生物自己会一代代发生变化。这种变化本来是没有方向的，生物自己也控制不了。变化是由什么导致的？是由父母的基因进行交换重组和基因突变导致的。在整个种群的不同个体中，有无数次基因重组的机会，也有一定基因突变的机会，这导致若干代以后，同一个种群中不同个体的样态的差别可能很大。看看现在的人类就知道了，人类有 23 对染色体，每一代的重组和突变使得人类的多样性特点非常明显，不仅在长相上有很大差别，在体态、性格、思维方式、疾病抵御能力等方面也千差万别。

尤其是人类历史上经历过的若干次大规模的瘟疫，那可都是杀人盈野的恶魔。14 世纪欧洲的黑死病，造成超过 7000 万人死亡。17 世纪在欧洲肆虐的天花病毒，也杀死了超过 4000 万名欧洲人。在 14 世纪得了黑死病（一种鼠疫杆菌），那基本没得救。防治天花的牛痘疫苗，也是 18 世纪才研发出来的，此前患病后存活下来的人，其实也没有接受像样的治疗。据科学家分析，这基本上只能解释为基因层面的胜利——患病后存活下来的人的基因，比那些因患病而死去的人的基因，包含更强的对抗这种疾病的成分。

不论是基因突变还是基因重组产生的新特性，都属于不确定的变化。而在客观世界中，很多条件都会对人类种群进行“裁剪”，疾病用于“裁剪”那些对疾病耐受力弱的人，饥荒用于“裁剪”那些对饥饿耐受力弱的人，严寒用于“裁剪”那些对寒冷耐受力弱的人——开个玩笑，就算有“颜值高”的基因也没用。残酷之处就在这里，没有对错，只有适应或者不适应。这种“进化”甚至谈不上“进化”，而是被动地演化，然后被动地“被选择”。

12.1.2 算法流程

在了解了基因进化层面的变化之后，还是要将这个过程落实到算法中。那么，我们怎么来构建这个算法的流程呢？其实，关键在于如下四步。

（1）基因编码

在基因编码过程中，我们尝试着描述一些个体的基因，构造这些基因的结构。这有点像确定函数自变量的过程。

（2）设计初始群体

在设计初始群体这个环节，我们需要在这个我们自己创造出来的小世界里“当家作主”，创造出一个种群。这些种群中有很多生物个体，它们的基因都不相同。

（3）适应度计算

在适应度计算（剪枝）环节，我们要创造一些“工具”，对种群中那些不太适应环境的个体进行“裁剪”，不让它们产生“后代”。在进行算法流程设计时，与那些我们最终的遴选规则差异较大的个体，肯定不适合作为备选对象，因此一定要将它们“裁剪”掉，否则，不仅由它们产生的“后代”会使计算量增大，而且对逼近目标没有任何帮助。

（4）产生下一代

有三种方法可以用于产生下一代：直接选择；基因重组；基因突变。回到上一步进行循环（适应度计算），产生下一代。这样一代一代找下去，直至找到最优解为止。

遗传算法在很多领域的应用中都体现出了很好的特性，例如旅行商问题（也叫作“货郎担问题”，Traveling Salesman Problem，TSP）、九宫格问题（还有八数码问题）、生产调度问题（Job Shop Scheduling）、背包问题（Knapsack Problem）等。

遗传算法的流程很简单，下面我们通过一个实例来更好地理解它。

12.1.3　背包问题

背包问题是一种组合优化的 NP 完全问题（Non-Deterministic Polynomial Complete Problem，多项式复杂程度的非确定性问题）。虽然读起来很拗口，但是特点很明显，那就是“生成问题的一个解通常比验证一个给定的解花费的时间要多得多”。

如果还是觉得不好理解，我们就来看几个例子。

一个例子是合数分解质因数问题。如果现在有一个命题：请将合数 698975355227 分解为几个质数相乘的结果。我们该怎么计算？只能把质数从 2 穷举到 $\sqrt{698975355227}$，然后 2 个质数一起、3 个质数一起、4 个质数一起……尝试各种组合相乘，看是否等于这个结果。即便我们有计算机，这也一项非常耗时的工作。但是，如果反过来说，698975355227 是否是 809、887、977、997 这四个素数的乘积，那么验证起来会非常快——只要我们手边有一个普通的计算器，半分钟都要不了就能验算出来。

我们在很多资料中都能看到对背包问题的描述，大意是：有 N 件物品和一个容量为 V 的背包，第 i 件物品的质量是 $w[i]$，价值是 $v[i]$，求解将哪些物品装入背包可使这些物品的质量总和不超过背包的容量且价值总和最大。这种问题就是典型的 NP 问题，即验证一个猜想的解比算出一组解要快得多。

通过一些数字可能会理解得更直观。假设有一个背包，其中可以放置 80 千克的物品。此外，还有 6 件物品，如表 12-1 所示。

表 12–1　6 件物品

物　品	质量（千克）	价　值
1	10	15
2	15	25
3	20	35
4	25	45
5	30	55
6	35	70

怎么放置才能让背包里的物品价值总和最大呢?

一种思路是穷举法。每种物品只有存在（1）和不存在（0）两种状态，那么可能产生的方案最多是 2^6 个（64 个）。把每种组合的质量和价值都算出来，超过 80 千克的方案就直接否定，不超过 80 千克的就留下并计算总价值，最后比较一下每种方案的总价值就可以了。这是一种可行的方法。

不过，如果有 128 件物品，这个方案还可行吗？如果使用穷举法，就是大约 3.4×10^{38} 种方案。如果计算机每秒能够验证 1 亿种情况，大概需要 1.08×10^{23} 年才能算出来——恐怕行不通。在这种情况下，遗传算法的优势就显现出来了。下面，我们看看在 6 件物品的情况下怎么求解。

1. 基因编码

一共 6 件物品，每件物品的有无都可以作为一个独立的基因片段，如表 12-2 所示。

表 12–2 基因片段

物 品 1	物 品 2	物 品 3	物 品 4	物 品 5	物 品 6	染 色 体
1	1	1	1	1	1	1111111

这个基因片段一共有 6 位。如果只有物品 2、物品 3 和物品 6，染色体是 011001。

2. 设计初始群体

为了计算方便，我们设置初始群体为 4 个随机产生的初始生物个体，用染色体来表示它们。

- 100100：对应于物品 1、物品 4 存在。
- 101010：对应于物品 1、物品 3、物品 5 存在。
- 010101：对应于物品 2、物品 4、物品 6 存在。
- 101011：对应于物品 1、物品 3、物品 5、物品 6 存在。

3. 适应度计算

适应度计算，需要一个适应度函数作为标尺。在本例中，适应度函数为物品总价值。那么，这 4 个个体的适应度的值分别是多少呢？具体如下。

- 100100：15+45=60。
- 101010：15+35+55=105。
- 010101：25+45+70=140。
- 101011：15+35+55+70=175。

不要忘了，计算结果不仅与物品本身有关，还与质量有关，所以质量也要作为判断标准之一。这 4 个个体的质量的值分别是多少呢？往下看。

- 100100：10+25=35。

- 101010：10+20+30=60。
- 010101：15+25+35=75。
- 101011：10+20+30+35=95。

染色体 101011 因为超过了我们要求的 80 千克而直接被淘汰。我们可以直接理解为，在人类的进化中，不知出于何种原因，一个人的体重飙升到 500 千克。然后呢？然后就没有“然后”了，即便这个人的智商再高，也很难生存了。

这样，剩下 3 个染色体，具体如下。

- 100100，适应函数的值为 60。
- 101010，适应函数的值为 105。
- 010101，适应函数的值为 140。

除了刚才那个体重飙升太严重的个体以外，其他个体会经历一个遴选的过程，更“健壮”的个体将有更多的机会生存下去。

我们先把适应函数的值求和，即 60+105+140=305，再进行一个类似“轮盘赌”的遴选过程，如图 12-1 所示。

图 12-1　轮盘

赌博我们肯定是不提倡的，这里只是用来打比方。“轮盘赌”是一种博彩游戏，游戏道具是一个可以旋转的木盘。木盘上有很多格子，每个格子上都有数字。轮盘开始旋转之后，将一个小球放在轮盘上，让它沿着盘面滚动（与轮盘的旋转方向相反）。待轮盘静止，小球所停靠的格子所对应的数字即为获胜号码（每个格子的胜率应该是相同的）。

现在我们想象一下：这个轮盘上有 305 个格子，染色体 100100 作为玩家选择了其中的 60 个格子，染色体 101010 作为玩家选择了其中的 105 个格子，染色体 010101 作为玩家选择了其中的 140 个格子，分别作为自己的“赌注”。轮盘转动 4 次——这个值推荐设置为每一代个体的数量（在这个小世界里，所有的事情我们说了算）。轮盘每转动一次，“中奖”的这个染色体就被允许繁殖一次。如果一次都没“中奖”，那真是太不幸了，这个基因将无法得到延续。

在这个例子里，我们会看到如下情况。

- 染色体 100100 每次被遴选的概率为 $\frac{60}{305}$。

- 染色体 101010 每次被遴选的概率为 $\frac{105}{305}$。
- 染色体 101010 每次被遴选的概率为 $\frac{140}{305}$。

在我的计算结果里，染色体 101010 和染色体 010101 各繁殖了两次。在你的模拟计算过程中可能不是这个结果。没关系，我们了解大体的思路就可以了。

4. 产生下一代

染色体 101010 和染色体 010101 在成功地被遴选后，需要通过基因重组来产生下一代。计算过程如表 12-3 所示。

表 12–3 产生下一代

个 体	染 色 体	配 对	交叉点位置	交叉结果
1	101010	1–2	3	101101
2	010101		3	010010
3	101010	3–4	4	101001
4	010101		4	010110

两个被遴选的基因进行了基因重组。一对从第 3 位后面断开，在尾部进行了交换；另一对从第 4 位后面断开，在尾部进行了交换。这样，就产生了 4 个不同的基因。一般来说，交叉点位置是可以随机选取的。如图 12-2 所示，两段不同的基因从中间断开后重新结合，上段的前半部和下段的后半部结合成为新的基因，下段的前半部和上段的后半部也结合成为新的基因。

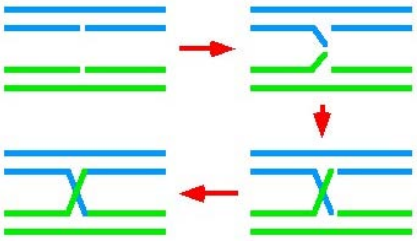


图 12-2 基因重组

在基因重组之后，可能有一个基因突变的过程，就是随机对一定比例的基因里的某一位或者某几位进行变化——1 变成 0，0 变成 1。对这个过程，建议还是取法一般的生物繁殖过程，让变异的基因比例低一些。在这个例子中没有变异发生。

5. 迭代计算

接下来，就是以这种准则做下去了。我们直接求质量和价值所对应的值，具体如下。

- 染色体 101101：质量 90，价值（适应函数）165——直接淘汰。

- 染色体 010010：质量 45，价值（适应函数）80。
- 染色体 101001：质量 65，价值（适应函数）120。
- 染色体 010110：质量 70，价值（适应函数）125。

在这里我们看到一个现象，总体的适应函数的和为 $80+120+125=325$，普遍比上一代的 $60+105+140=305$ 的适应性好——貌似进化了。但是，上一代有一个适应函数 140 的“超强基因个体”，这一代却没能“超”过去。

在一次完整的计算中，迭代过程可能会经历几十代甚至更久，如果发现出现了连续几代适应函数基本不增加或者减少的情况，就说明函数已经收敛了。

如果没有在算法的学习中接触过“收敛”这个词，可以通过一个形象的例子来理解。想象一下，我们平时在体重秤上称量的时候，人站到秤上，指针（或者显示的数值）就开始抖动（变化），且抖动的幅度越来越小，最后基本稳定在一个值。数值稳定后，我们可以读取数字了。如果体重秤的称量是由算法控制的，那么抖动几次很快就能稳定在一个值，这就是一个收敛比较快（比较好）的算法；如果数值要抖动很久才能稳定下来，这就是一个收敛比较慢（比较差）的算法；如果抖动的幅度随着时间的推移越来越大，这就是一个收敛性非常不好的算法，通常无法得到解。

在本节的例子里，既可以就此结束迭代操作，也可以再观察一到两代的变化。收敛的速度会因为很多因素而变化，例如基因位的长度、基因重组时的方案、基因变异的程度、每一代产生个体的数量等。发生适应函数收敛的时候通常就是迭代结束的时候，而在迭代结束前找到的最优解就是我们想要的解。

6. 注意事项

在使用遗传算法的时候，如下方面是可以进行调整的。

（1）初始群体

初始群体的数量是可以调整的。在本节的 6 件物品的背包问题中，极限是我们直接生成所有的情况，即 2^6 个个体全部作为初始群体被列出的情况。但是，这不仅毫无意义，也不是我们使用基因算法的目。一般来说，我们不会用遗传算法解决复杂度这么低的问题。在对群体进行初始化的时候，数量要多而丰富，才能让基因之间的重组与淘汰更为有效。

（2）适应度函数

在适应度函数中，“轮盘赌”算法只是其中之一，也可以考虑使用其他算法进行遴选。需要注意的是，我们的遴选原则，其实就是从多样化的生物中进行挑选，所以，可以淘汰一些比较“弱”的基因，但淘汰的比例不要太大。

（3）基因重组

基因重组这个环节的变数是比较大的，断开的位置几乎可以是任意的。在本节的例子中，对

一个 6 位的染色体，1–5 断开、2–4 断开、3–3 断开、4–2 断开、5–1 断开都是可以选择的方案。其实，在一次产生后代的过程中，允许以不同的方案产生多个后代。例如，两个配对的基因可以用 2–4 断开方案产生两个后代，同时用 4–2 断开方案产生两个后代，即产生 4 个后代。这样会使基因变得更丰富。不过，同时要注意计算量。如果计算量增加了，在若干代以后，恐怕会严重影响计算性能。

另外，相信大家也能意识到，不要让一个基因自身和自身重组，这是没有意义的。自己无论怎么和自己重组，得到的还是自己——“近亲结婚”害处大，至少不会得到更新、更好的方案。

（4）迭代结束

算法迭代结束的判断标准因人而异，但总体来说就是，如果连续几代都没有明显的适应函数增长，就说明进化基本“到头”了。当结束迭代时，在此之前找到的最优解，就是我们能找到的最优解。

对于本节中的背包问题，也给出了一段 Python 代码（在本书 GitHub 页面 ~/DRLPytorch/ga 目录下的 bag.py 中）。

```
import random
#背包问题
#物品质量和价格
X = {
    1: [10, 15],
    2: [15, 25],
    3: [20, 35],
    4: [25, 45],
    5: [30, 55],
    6: [35, 70]}

#终止界限
FINISHED_LIMIT = 5

#质量界限
WEIGHT_LIMIT = 80

#染色体长度
CHROMOSOME_SIZE = 6

#遴选次数
SELECT_NUMBER = 4

max_last = 0
diff_last = 10000

#收敛条件、判断退出
def is_finished(fitnesses):
    global max_last
```

```
    global diff_last

    max_current = 0
    for v in fitnesses:
        if v[1] > max_current:
            max_current = v[1]

    diff = max_current - max_last
    if diff < FINISHED_LIMIT and diff_last < FINISHED_LIMIT:
        return True
    else:
        diff_last = diff
        max_last = max_current
        return False

#初始染色体样态
def init():
    chromosome_state1 = '100100'
    chromosome_state2 = '101010'
    chromosome_state3 = '010101'
    chromosome_state4 = '101011'
    chromosome_states = [chromosome_state1,
                         chromosome_state2,
                         chromosome_state3,
                         chromosome_state4]
    return chromosome_states

#计算适应度
def fitness(chromosome_states):
    fitnesses = []
    for chromosome_state in chromosome_states:
        value_sum = 0
        weight_sum = 0
        for i, v in enumerate(chromosome_state):
            if int(v) == 1:
                weight_sum += X[i + 1][0]
                value_sum += X[i + 1][1]
        fitnesses.append([value_sum, weight_sum])
    return fitnesses

#筛选
def filter(chromosome_states, fitnesses):
    #质量大于 80 的被淘汰
    index = len(fitnesses) - 1
    while index >= 0:
        index -= 1
        if fitnesses[index][1] > WEIGHT_LIMIT:
            chromosome_states.pop(index)
            fitnesses.pop(index)
```

```
 79
 80     #遴选
 81     selected_index = [0] * len(chromosome_states)
 82     for i in range(SELECT_NUMBER):
 83         j = chromosome_states.index(random.choice(chromosome_states))
 84         selected_index[j] += 1
 85     return selected_index
 86
 87 #产生下一代
 88 def crossover(chromosome_states, selected_index):
 89     chromosome_states_new = []
 90     index = len(chromosome_states) - 1
 91     while index >= 0:
 92         index -= 1
 93         chromosome_state = chromosome_states.pop(index)
 94         for i in range(selected_index[index]):
 95             chromosome_state_x = random.choice(chromosome_states)
 96             pos = random.choice(range(1, CHROMOSOME_SIZE - 1))
 97             chromosome_states_new.append(chromosome_state[:pos] +
chromosome_state_x[pos:])
 98         chromosome_states.insert(index, chromosome_state)
 99     return chromosome_states_new
100
101 if __name__ == '__main__':
#初始群体
103     chromosome_states = init()
104     n = 100
105     while n > 0:
106         n -= 1
107         #适应度计算
108         fitnesses = fitness(chromosome_states)
109         if is_finished(fitnesses):
110             break
111
112         #遴选
113         selected_index = filter(chromosome_states, fitnesses)
114
115         #产生下一代
116         chromosome_states = crossover(chromosome_states, selected_index)
117     print(chromosome_states)
118
119 # 1: [[60, 35], [105, 60], [140, 75], [175, 95]]
120 # 2: [[60, 35], [105, 60], [80, 45], [90, 50]]
121 # 3: [[95, 55], [115, 65], [70, 40], [90, 50]]
122 # 4: [[70, 40], [70, 40], [150, 85], [115, 65]]
123 # 5: [[115, 65], [115, 65], [115, 65], [70, 40]]
124 # ['100110', '100110', '100110', '100110']
```

我们使用的收敛条件是：连续两代的适应函数最大值不再增加。得到的结论是：[115, 65] 是我们要求的解，对应的是物品 1（[10, 15]）、物品 4（[25, 45]）、物品 5（[30, 55]）。

12.1.4　极大（小）值问题

在本节中，举一个求极大值的例子。这种例子是很常见的，只要我们把一些数据关系描述成函数，就会产生一些求极大值或者极小值的问题。其实，极大值和极小值是一类问题，就是极值问题，解题思路也一般无二。

虽然可以使用梯度下降法来解决极值问题，但对于那些褶皱非常的“多峰函数”，该怎么处理呢？我们一起看一个早就信手拈来的函数。

假设在空间里有一个函数 $z = y\sin(x) + x\cos(y)$，如图 12-3 所示。

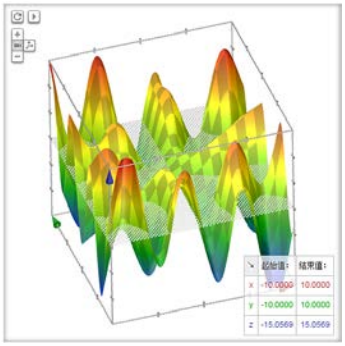


图 12-3　函数 $z = y\sin(x) + x\cos(y)$（1）

找不到这个函数所对应的物理解释也没关系，我们把它当成丘陵地区的地形图就好了。每个点的 (x, y, z) 三维坐标都可以用

$$z = y\sin(x) + x\cos(y)$$

来描述。

如果我们想求这个函数在 x 位于 $[-10,10]$、y 位于 $[-10,10]$ 之间的最大值，该怎么办呢？对于微积分学得比较好的读者朋友，这根本不是问题。

把 z 对 y 求偏导数，把 z 对 x 求偏导数，这样，就能够求出满足两个偏导数同时为 0 的 x、y 了。解很可能有多个。把所有的 x 和所对应的 y 代入 $z = y\sin(x) + x\cos(y)$，求出的 (x, y, z) 就是驻点。比较每个驻点的 z 的值，最大的那个就是要求的解。不过，本节要介绍的不是这种微积分领域的常用方法，我们还是考虑用迭代法中的遗传算法的思路来解决问题。

在这个层峦叠嶂的小小区域，一切还是由我们说了算。

可以这么想象：在这个地区，有我们创造的很多个无规律的个体，它们有的在谷底，有的在半山腰，有的已经在山顶或者山顶附近。那么，让它们一代一代地繁殖，凡是能爬得更高的就留

下来，至于爬不上去的——就在另一个“平行世界”里生活吧，我们就不观察了。

有了这个思路，我们就还是按部就班地做做看。

1. 基因编码

在这个例子中，我们遇到的基因编码问题和背包问题不一样。

刚刚我们说的背包问题属于离散型的自变量，一个物品在背包里，要么有、要么没有，即便是 128 个物品，我们也知道一共有 2^{128} 种情况，基因编码最多用 128bit 就能表示了。

但是在这个例子中，有两个自变量 x、y。这两个自变量都是实数。根据实数稠密性原理，在 $[-10,10]$ 中有无穷个实数，因此，我们没有办法用类似背包问题中的基因编码的方法穷举所有的实数。

不要惊慌。不知道大家注意过没有，计算机在做所有实数运算的时候，其实得到的结果不是特别精确。因为用电子计算机来计算实数，就是一个用离散解决连续、用有穷解决无穷的方案，本身就有天生的、不可逾越的局限性。不管是用 64bit 来描述一个实数，还是用 128bit 来描述一个实数，都太有限了。所以，我们需要一个精度限制，也就是我们平时说的有效数字。问题是，应该取多少位有效数字？

在理想情况下，有效数字应该尽可能多取，有多少就取多少。但是在实际生活中，我们却不会这么做。原因也很简单，取多位有效数字本来就是一个提高精确度、降低误差与成本的过程。然而，取多位有效数字需要更高的成本，如果有效数字位数的增长对提高收益没有明显的好处，那显然是不划算的。

我们平时在购物时，金额取小数点后 2 位就够了，因为再小的货币单位没有发行，更小的标价单位毫无意义。而对一些大宗交易的物品，例如电容、放大器之类的电子元件，其单价的小数点后位数通常会取得比较多，很多元件的出厂单价只有 0.005 元甚至更低。由于购买量巨大，一批就可能有上千万个，取 4 位小数就是有意义的。再如，生产 CPU 时使用的高纯硅的纯度通常是 99.999999999%（11 个 9），纯度低了就无法支持 22 纳米级别的生产工艺，而太阳能板硅板就不用这么精细了，一般“4 个 9”就能满足，如果超过“6 个 9”，只会造成成本的增加。

所以，对于那些我们无法精确感知的、即便感知了也无法把控的、即便把控了也对提高效益无意义的场景，就没必要提升精度了。在生产和生活中，大家酌情取舍就好了。

回头来看本节中的例子。如果允许一定的误差存在，那么只要误差足够小，就能够满足模型的需求了。

假设 $[-10,10]$ 表示 20 千米的距离，精度为 1 米就可以理解为在 $[-10.000,10.000]$ 的范围内取值，精度为 0.1 米就可以理解为在 $[-10.0000,10.0000]$ 的范围内取值。我们假设 1 米的精度已经能够满足误差要求了，以 $[-10.000,10.000]$ 作为取值范围。这样，“无限”就变成了“有限”，实际有 20001 种取值。

如果我们还是要用二进制来进行基因编码，应该选用多少位呢？应该有 2^{14}（16384）、2^{15}

（32768）。对于这 20001 种情形，就取 15bit 作为基因描述吧。x 和 y 都用 15bit 的基因，这已经能够覆盖所有的设想样本了。

下一个问题，就是把一个 15bit 的二进制数字映射到 $[-10.000,10.000]$ 的区间里去。难吗？一点都不难。我们用第一个 bit 作为正负数的标识，将剩下的 14bit 作为具体数字的标识，这样，这个 15bit 的数字就相当于对 $[-16383,0]$ 和 $[0,16383]$ 两个区间取并集了（集合里的元素都是整数）。再把每个数字除以 16383，然后除以 1000，取值范围就被压缩在 $[-10.000,10.000]$ 中了。

为了让区间包括两边的边界点，我们做一个小小的变换。定义 $F(x)$ 的内容如下：x 是自变量，二进制数字，首位为 0 代表正数，首位为 1 代表负数；函数值为 x 所对应的十进制数字。那么，我们就完成了编码到整个定义区间上的映射 $F(x)$。最后，表示二进制到实数值的函数 $H(x)$ 有如下定义：

$$H(x)=\begin{cases}10\cdot\dfrac{F(x)+1}{16384} & F(x)\geqslant 0\\ 10\cdot\dfrac{F(x)-1}{16384} & F(x)\leqslant 0\end{cases}.$$

细心的读者朋友可能会看出，这个地方有问题了——有两个问题。

- 问题 1：$F(x)\geqslant 0$ 和 $F(x)\leqslant 0$ 是有重叠部分的。$F(x)=0$ 究竟算谁的？那就补充说明一下吧。这个写法确实不够严谨。具体意义是 *x* 的首位为 0 还是为 1 的情况：如果为 0，那么 $F(x)\geqslant 0$；如果为 1，那么 $F(x)\leqslant 0$。在后面的 14bit 都为 0 的情况下，不管首位为 0 还是为 1，值都是 0。
- 问题 2：$10\cdot\frac{F(x)+1}{16384}$ 和 $10\cdot\frac{F(x)-1}{16384}$ 分别构成了 *x* 的正、负两部分，但是区间却不是我们原来说的 $[-10.000,10.000]$，变成了 $[-10.000,0)$ 和 $(0,10.000]$，0 从自变量范围里被拿掉了。我承认，这只是为了计算方便而设计的。不过，我确信 0 不是我们要的解（不管是 $x=0$，还是 $y=0$），所以，把解里面所有 $x=0$ 的情况及所有 $y=0$ 的情况都去除了，这个变换不影响最终求解。如果不放心，可以尝试重新构造这个映射关系，覆盖 $[-10.000,10.000]$ 中的所有点。

最后，用距离说明映射关系的具体值。

- $(000000100100100)_2=292$，对应于 $10\cdot\frac{F(x)+1}{16384}=0.179$。
- $(111000100110110)_2=-12598$，对应于 $10\cdot\frac{F(x)-1}{16384}=-7.690$。

怎么样，挺容易的吧？用现在流行的词来说——完美。

2. 设计初始群体

什么都自己说了算不是一件容易的事情，设计初始群体就是我们要解决的第二个大问题。

换个角度（倾斜一下）看看我们的小世界，如图 12-4 所示。我们要设置一些初始化的个体，让它们一代一代繁衍，对能爬得更高的继续观察，对爬不高的就不再管了。

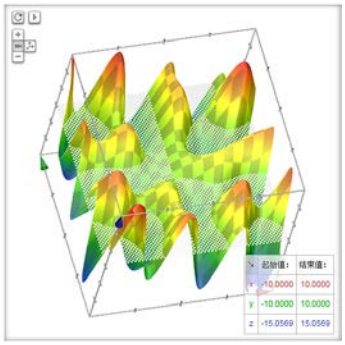


图 12-4　函数 $z = y\sin(x) + x\cos(y)$（2）

在我们设计的基因组里其实有两条基因，一条是 x，另一条是 y。这两条基因各有 15 个基因信息点，也就是 2^{15} 个可能性。我们随机产生 8 组，如表 12-4 所示。

表 12–4　随机产生 8 组基因

染 色 体	基因 X	基因 Y
1	000000100101001	101010101010101
2	011000100101100	001100110011001
3	001000100100101	101010101010101
4	000110100100100	110011001100110
5	100000100100101	101010101010101
6	101000100100100	111100001111000
7	101010100110100	101010101010101
8	100110101101000	000011110000111

3. 适应度计算

适应度计算的设计在这个场景里不难，用 $z = y\sin(x) + x\cos(y)$ 即可（z 就是适应度）。

4. 产生下一代

每一代都可以让一个染色体中的基因 X 和基因 Y 进行组合，如表 12-5 所示。

表 12–5

染 色 体	基因 X	基因 Y	组　　合	后代基因
1	000000100101001	101010101010101	1X+2X	XA: 000000100101100
				YA: 101010110011001
2	011000100101100	001100110011001	1Y+2Y	XB: 011000100101001
				YB: 001100101010101

如果染色体 1 和染色体 2 结合，那么有：

- 染色体 1 的 X 基因的前 7 位和染色体 2 的 X 基因的后 8 位结合；
- 染色体 1 的 Y 基因的前 7 位和染色体 2 的 Y 基因的后 8 位结合；
- 染色体 2 的 X 基因的前 7 位和染色体 1 的 X 基因的后 8 位结合；
- 染色体 2 的 Y 基因的前 7 位和染色体 1 的 Y 基因的后 8 位结合。

由此形成 XA、YA、XB、YB 四个后代基因，XA 和 YA 将成为一组新的染色体，XB 和 YB 将成为一组新的染色体，从而形成两个完整的后代基因染色体组。

如果以 8 组作为初始种群，就有 $C_8^2 = 28$ 种组合方式，每一种组合产生 2 个后代，那么实际上第一代以后就会产生 56 个个体。

这 56 个个体的适应度，我们可以拿来排序，只取出排名前 8 位的个体。

在这里同样可以允许一定的基因突变性。在 8 个遴选出来的个体中，随机找到 2 个，让这 2 个个体中的一个的 X 染色体和另一个的 Y 染色体发生变异——巧了，由 x 和 y 两个自变量构成的染色体和人类的性染色体同名了。在这里，变异就是随机改变 X 染色体或者 Y 染色体中的某一位。

此后，再进行两两重组的计算，产生下一代。

在这里要注意如下两点。

（1）断开点的位置

理论上，断开点的位置是任意的，只不过——断开点靠左对数值的影响大，自变量“跳跃”的范围就大；断开点靠右对数值的影响小，自变量“跳跃”的范围就小。

（2）基因变异的位置

基因变异的位置和断开点的位置产生的影响是完全一样的——变异点靠左，自变量“跳跃”范围大；变异点靠右，自变量“跳跃”范围小。

5. 迭代计算

迭代计算的过程，我们直接看代码就好。因为有很多随机因素存在，所以这里展示的计算过程可能会和读者看到的不一致，迭代的“代”数也可能不一致，但结果应该是一样的。

代码具体如下。

```
import random
import math
import numpy as np

#染色体长度
CHROMOSOME_SIZE = 15

#判断退出
```

```
def is_finished(last_three):
    s = sorted(last_three)
    if s[0] and s[2] - s[0] < 0.01 * s[0]:
        return True
    else:
        return False

#初始染色体样态
def init():
    chromosome_state1 = ['0000000100101001', '101010101010101']
    chromosome_state2 = ['0110000100101100', '0011001100011001']
    chromosome_state3 = ['0010001001001001', '1010101010101101']
    chromosome_state4 = ['0001101001001001', '1100110011001101']
    chromosome_state5 = ['1000001001001001', '1010101010101101']
    chromosome_state6 = ['1010001001001001', '1111000011110001']
    chromosome_state7 = ['1010101001101001', '1010101010101101']
    chromosome_state8 = ['1001101011010001', '0000111100001111']
    chromosome_states = [chromosome_state1,
                         chromosome_state2,
                         chromosome_state3,
                         chromosome_state4,
                         chromosome_state5,
                         chromosome_state6,
                         chromosome_state7,
                         chromosome_state8]
    return chromosome_states

#计算适应度
def fitness(chromosome_states):
    fitnesses = []
    for chromosome_state in chromosome_states:
        if chromosome_state[0][0] == '1':
            x = 10 * (-float(int(chromosome_state[0][1:], 2) - 1)/16384)
        else:
            x = 10 * (float(int(chromosome_state[0], 2) + 1)/16384)
        if chromosome_state[1][0] == '1':
            y = 10 * (-float(int(chromosome_state[1][1:], 2) - 1)/16384)
        else:
            y = 10 * (float(int(chromosome_state[1], 2) + 1)/16384)
        z = y * math.sin(x) + x * math.cos(y)

        fitnesses.append(z)

    return fitnesses

#筛选
def filter(chromosome_states, fitnesses):
    #top 8 对应的索引值
    chromosome_states_new = []
```

```
    top1_fitness_index = 0
    for i in np.argsort(fitnesses)[::-1][:8].tolist():
        chromosome_states_new.append(chromosome_states[i])
        top1_fitness_index = i
    return chromosome_states_new, top1_fitness_index

#产生下一代
def crossover(chromosome_states):
    chromosome_states_new = []
    while chromosome_states:
        chromosome_state = chromosome_states.pop(0)
        for v in chromosome_states:
            pos = random.choice(range(8, CHROMOSOME_SIZE - 1))
            chromosome_states_new.append([chromosome_state[0][:pos] + v[0][pos:],
chromosome_state[1][:pos] + v[1][pos    :]])
            chromosome_states_new.append([v[0][:pos] + chromosome_state[1][pos:],
v[0][:pos] + chromosome_state[1][pos    :]])
    return chromosome_states_new

#基因突变
def mutation(chromosome_states):
    n = int(5.0 / 100 * len(chromosome_states))
    while n > 0:
        n -= 1
        chromosome_state = random.choice(chromosome_states)
        index = chromosome_states.index(chromosome_state)
        pos = random.choice(range(len(chromosome_state)))
        x = chromosome_state[0][:pos] + str(int(not
int(chromosome_state[0][pos]))) + chromosome_state[0][pos+1:]
        y = chromosome_state[1][:pos] + str(int(not
int(chromosome_state[1][pos]))) + chromosome_state[1][pos+1:]
        chromosome_states[index] = [x, y]

if __name__ == '__main__':
    chromosome_states = init()
    last_three = [0] * 3
    last_num = 0
    n = 100
    while n > 0:
        n -= 1
        chromosome_states = crossover(chromosome_states)
        mutation(chromosome_states)
        fitnesses = fitness(chromosome_states)
        chromosome_states, top1_fitness_index = filter(chromosome_states,
fitnesses)
        last_three[last_num] = fitnesses[top1_fitness_index]
        if is_finished(last_three):
            break
        if last_num >= 2:
```

```
            last_num = 0
        else:
            last_num += 1

    print(chromosome_states)
#x: 7.69897460938  y:7.69897460938  z:8.79528923825
#x: -8.35693359375  y:-8.35693359375 z:11.3501994249
```

最后的收敛条件是：判断每一代的适应函数最大值并做记录；判断最近三代的最大值如何变化。如果最近三代的适应函数最大值相比，第一大的（最大的）比第三大的（最小的）增益，值小于 1%，就判断为收敛。这种收敛的速度非常快，大概三到四代就可以收敛完毕，求出最优解。当然，对这个部分，我们同样可以采用更为考究的计算方式——作为对此感兴趣的读者朋友继续研究的一点提示好了。

需要注意，在执行这段代码时可能会发现一个问题：如果 z 的值收敛到 8.8 附近，x 和 y 的值都在 7.7 附近（就像我们在例子中给出的）；如果 z 的值收敛到 11.35 附近，x 和 y 的值都在 –8.35 附近。根据这两种不同的解，我们就可以知道，后者应该是正确的——绝非前者。

如图 12-5 所示，用圆圈圈住的位置就是 $(7.7, 7.7)$ 附近的点，而实际上，正确的解在方框框住的位置 $(-8.35, -8.35)$。为什么会出现这种现象呢？就是因为在繁殖下一代的时候又出现了一次“播撒”的情况。但由于“播撒”不够均匀，导致部分爬到“圆圈山”位置的种群个体表现格外好，而其他种群，例如在“方框山”半山腰的这些，就没有被遴选出来——从第 9 名开始都会被淘汰。这样，就有相当的概率会收敛到 8.8 附近的“圆圈山”。这类问题，我们在编写遗传算法的过程中也会遇到，请读者朋友注意。

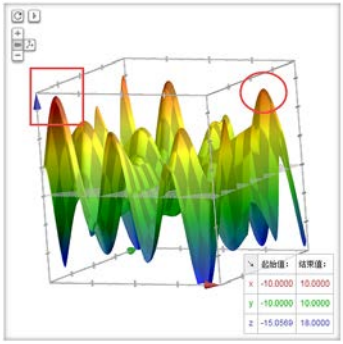


图 12-5　函数$z = y\sin(x) + x\cos(y)$（3）

怎么破解这个问题呢？我觉得可以考虑以下两个方法。

（1）初始种群扩大化

初始种群中的个体数量可以不是 8 个，而是 16 个、32 个甚至更多。总之，就是在一开始就使得第一代有足够的机会爬到“方框山”的较高位置。

（2）增加每一代的遴选名额

现在，我们只留下了种群中每一代的 8 个个体。同样，我们可以留下 16 个、32 个甚至更多，这会让更多爬到“方框山”较高位置的个体存活下来。

同时采用这两种方法，找到最优解的概率会大大增加。

12.2　NEAT 原理

在了解了遗传算法的基本原理和用法之后，我们就可以回过头来看一下 NEAT 了。

最早的论述 NEAT 的论文是 *Efficient Evolution of Neural Network Topologies*（http://nn.cs.utexas.edu/downloads/papers/stanley.cec02.pdf）。该论文发表于 2002 年，仅有 6 页 A4 纸的内容，看上去干货满满。

另一篇论文 *Evolving Neural Networks through Augmenting Topologies*，在 2002 年被 *The MIT Press Journals* 收录。这篇论文也可以在网上找到，地址为 http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf。这篇文章的篇幅比较长，有 30 页 A4 纸，内容与前一篇论文相近，不过其中的铺垫和细节更多，如果有时间可以仔细阅读一下。在本节中，我们重点讨论其中的原理。

前面我们反复讨论的深度学习与强化学习相结合的方法（DQN 就是一个 Q-Learning 和 CNN 解决回归问题的工程结合手段），说老实话，并没有太多令人眼前一亮的创新内容。Q-Learning 历史悠久自不必说，用 CNN 解决回归问题其实也是老生常谈，就是利用卷积核的特征提取和神经网络丰富的 VC 维来解决传统机器学习不容易解决的复杂的线性不可分问题。

NEAT 在理解一个从输入 S 到输出 A 之间关系的神经元组合时，认为这是一个 NP 问题，就像那个在一个背包里放一些物品，保证背包不被撑破并且物品价值尽可能大的问题一样。既然是 NP 问题，就完全可以考虑使用像遗传算法这样的方法来一代一代实现基因的优胜劣汰，然后使基因所代表的神经元组合的解逐渐向着满足尽可能高的适应度的方向进化。这就是 NEAT 最为基本的思路了。

12.2.1　基因组

既然从形式上看，一个网络的描述变成了像基因这样的描述，那么它的形式就与之前我们见到的一般的 CNN 和 FF 网络有所不同，如图 12-6 所示。

需要注意的是，在 NEAT 的世界里，连接通常都显得不那么规则。例如，如图 12-6 所示的这种形式，节点 1 和节点 2 作为节点 4 的输入，而右侧的节点 3 可以直接输入后面的节点 5。这种不对称、不规则的形式，是 NEAT 的一个特点。

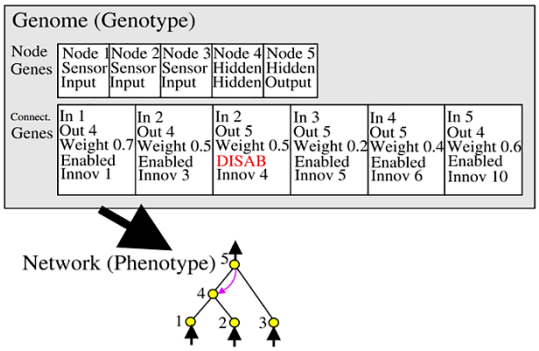


图 12-6　NEAT 用基因来描述网络连接

一个完整的基因组（Genome）中有两类节点描述，一类是节点基因（Node Genes），另一类是连接基因（Connect Genes）。重点看图 12-6 中的连接基因，一个连接基因携带的信息如下。

- 输入节点：In。
- 输出节点：Out。
- 权重：Weight。
- 有效性：Enabled。第 3 个基因是“DISAB”，其实就是“Disable”（无效）的意思，因此，其下面就没有从节点 2 到节点 5 可以连通的情况了（其他几个 Enabled 节点都是从节点 2 到节点 5 可连通的）。
- 创新号：Innov。

这些信息就可以描述网络中各节点之间的关系了。一张网络拓扑图可以通过有限的基因信息描述将网络完整地复现。

12.2.2　变异和遗传

在 NEAT 中，当然少不了介绍另外两个关键的环节，一个是变异（如图 12-7 所示），另一个是遗传（如图 12-8 所示）。

论文中提到的变异有如下两种。

- 第一种是添加连接的变异，就是在已有的连接节点中随机加入一个新的连接。如图 12-7 所示，在上半部分“Mutate Add Connection”的节点 3 到节点 4 之间，直接添加了这样一个新的连接。
- 第二种是添加节点的变异，就是增加一个新的连接节点（节点基因），并随机添加一些连接关系（连接基因），因为仅添加一个连接节点而不添加连接关系是没有意义的。如图 12-7 所示，“Mutate Add Node”部分就表示了节点的变异过程。

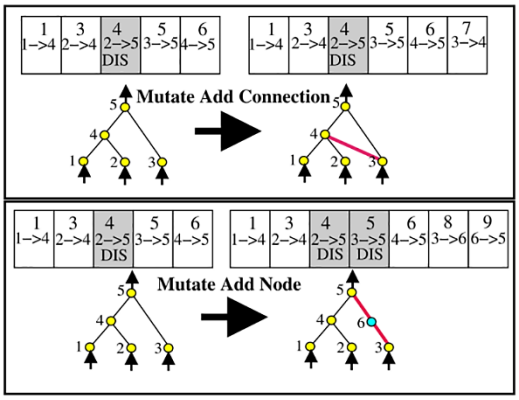


图 12-7　NEAT 的变异

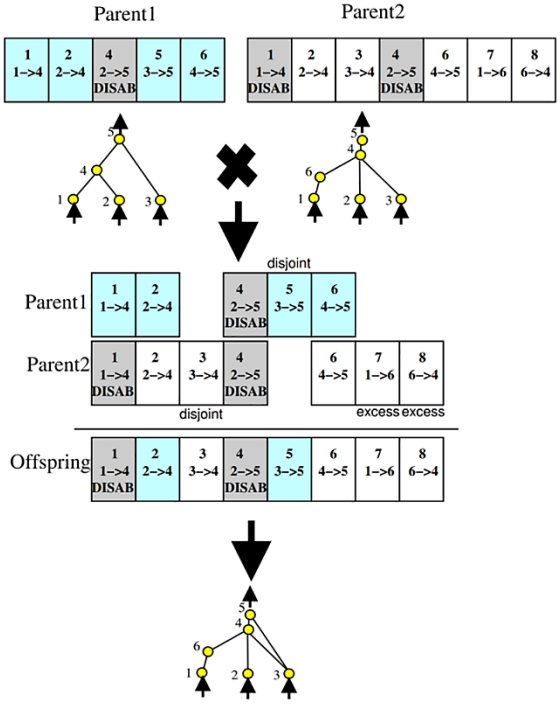


图 12-8　NEAT 的遗传

简单介绍一下 NEAT 遗传的过程。

首先，对“双亲”的基因进行对齐操作，也就是把它们的 Innov 两两对齐。对齐后，如果双方都有该 Innov，就随机选择一个保留。如果出现 Innov 不匹配的基因，例如图 12-8 中的 3 号基因和 7 号、8 号基因，在产生后代的时候，会由后代继承适应度（Fitness）较高的一方的基因。在这

里，应该假设“双亲”Parent1 和 Parent2 的适应度一样高，此时会随机继承一方的基因（也可以说，选择继承或者不继承）。

每按照这样的步骤进化一代，就在环境中用新的基因组网络做一次测试，也就是在环境中真正地运行若干个 Epoch，并计算 Fitness，就能够完成整个任务了。这与背包问题和求极值问题的解决方式如出一辙。

12.3 NEAT 示例

既然 NEAT 这么好用，我们赶紧拿来做个实验吧。

网上有一篇流传很广的用 NEAT 实现任天堂《超级马里奥兄弟》游戏自我进化的文章，我曾经亲自测试过，效果确实不错，如图 12-9 所示。

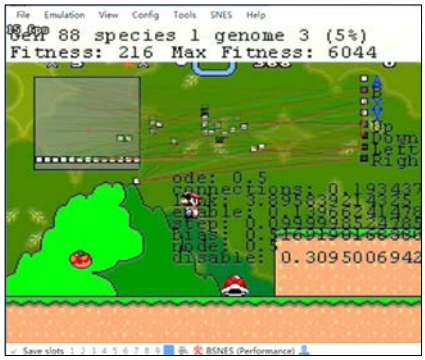


图 12-9 用 NEAT 玩《超级马里奥兄弟》

不过，这个 Case 作为本书的例子不太合适。

一是，这个 Case 依赖 NES（Nintendo Entertainment System）模拟器。这类模拟器的成熟运行环境主要在 Windows 上，所以，要做这个实验，就要把系统换回 Windows。为了做一个实验而更换操作系统，似乎没有必要。

二是，这个《超级马里奥兄弟》的 NES 模拟器确实可以从网上下载，其中包括《超级马里奥兄弟》的软件包，以及完成这个功能的脚本 neat.lua。但是，这些都是基于 Lua 语言实现的，而且没有进行封装。代码读起来确实痛快，但其迁移性比较差。Lua 本身就不是机器学习的主流语言环境，为了读一个脚本多学一门语言，意义也不大。此外，NEAT 有比较成熟的 Python 软件包 neat-python，直接用 pip 安装非常方便，我们不如直接用这个包来实现各种实验。

输入如下命令，直接安装 neat-python 软件包。

```
pip3 install neat-python
```

一下子就安装好了。

下面我们来看一个小实验——Cartpole。

Cartpole 是 Gym 环境中的一个小游戏，它模拟了用小车立杆子的环境，如图 12-10 所示。

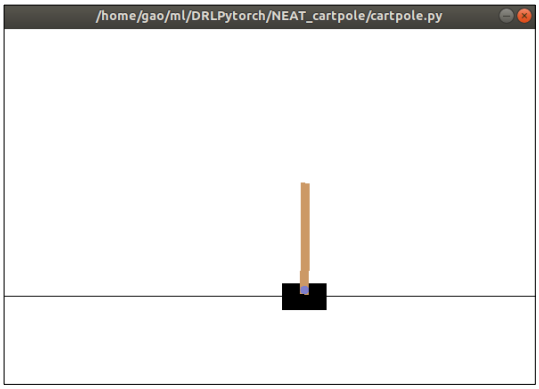


图 12-10　用 NEAT 玩 Cartpole 游戏

程序并不长，我们就不逐行解释了。

在这个小游戏中，输入的内容包括 In0 ~ In3 共 4 个值，如果打印出来，大概是这样的：

```
[ 0.02259702  0.02328209  -0.00097078  -0.03043893]
[ 0.01962615  0.02331854   0.00365919  -0.03124317]
```

它们各自的含义我还没有具体研究过。不过，无论是描述杆子两个端点的相对位置，还是一个端点的位置，甚至杆子的角度，4 个维度值已经足够了。

输出的内容是一个方向，一个是 `act1`，另一个是 `act2`，它们分别代表左和右。当网络正常传播时，也就是 `net.actives(s)`工作时，会输出一个二维向量 `[act1, act2]`。用 `np.argmax()`对它进行操作，相当于哪个维度的值大，就输出哪个维度的下标值。

- 当取 `[0.7813504261444919, 0.636593331028009]` 时，最终输出动作为 0。
- 当取 `[1.2234539393760717, 1.2623265641303656]` 时，最终输出动作为 1。

程序代码只有不到 80 行，具体如下。

```
 1 import neat
 2 import sys
 3 import numpy as np
 4 import gym
 5 import visualize
 6
 7 GAME = 'CartPole-v0'
 8 env = gym.make(GAME).unwrapped
 9
10 CONFIG = "./config"
11 EP_STEP = 300
```

```
GENERATION_EP = 10
CHECKPOINT = 9

def eval_genomes(genomes, config):
    for genome_id, genome in genomes:
        net = neat.nn.FeedForwardNetwork.create(genome, config)
        ep_r = []
        for ep in range(GENERATION_EP):
            accumulative_r = 0
            observation = env.reset()
            for t in range(EP_STEP):
                action_values = net.activate(observation)
                action = np.argmax(action_values)
                observation_, reward, done, _ = env.step(action)
                accumulative_r += reward
                if done:
                    break
                observation = observation_
            ep_r.append(accumulative_r)
        genome.fitness = np.min(ep_r)/float(EP_STEP)

def run():
    config = neat.Config(neat.DefaultGenome, neat.DefaultReproduction,
                         neat.DefaultSpeciesSet, neat.DefaultStagnation, CONFIG)
    pop = neat.Population(config)

    # recode history
    stats = neat.StatisticsReporter()
    pop.add_reporter(stats)
    pop.add_reporter(neat.StdOutReporter(True))
    pop.add_reporter(neat.Checkpointer(5))

    pop.run(eval_genomes, 10)

    # visualize training
    visualize.plot_stats(stats, ylog=False, view=True)
    visualize.plot_species(stats, view=True)

def evaluation():
    p = neat.Checkpointer.restore_checkpoint('neat-checkpoint-%i' % CHECKPOINT)
    winner = p.run(eval_genomes, 1)    # find the winner in restored population

    # show winner net
    node_names = {-1: 'In0', -2: 'In1', -3: 'In3', -4: 'In4', 0: 'act1', 1:
'act2'}
    visualize.draw_net(p.config, winner, True, node_names=node_names)

    net = neat.nn.FeedForwardNetwork.create(winner, p.config)
    while True:
```

```
60        s = env.reset()
61        while True:
62            env.render()
63            a = np.argmax(net.activate(s))
64            s, r, done, _ = env.step(a)
65            if done: break
66
67 if __name__ == '__main__':
68    TRAINING = sys.argv[1]
69
70    if TRAINING == 'TRAIN':
71        run()
72    elif TRAINING == 'EVAL':
73        evaluation()
74    else:
75        print('Please indicate TRAIN or EVAL')
```

在训练的时候，执行如下命令。

```
python cartpole.py TRAIN
```

整个训练过程非常快，即便使用 CPU，也只需要几十秒。

训练结束后，使用如下命令进行测试。

```
python cartpole.py EVAL
```

在测试的时候，会调用 visualize 组件来绘制整个网络的拓扑结构图。

NEAT 算法本身的特性，会导致每次训练的结果区别非常大。我连续训练了两次，结果就很不一样。用 NEAT 玩 Cartpole 小游戏得到的网络拓扑图，如图 12-14 所示。

NEAT 还有一个特点，就是网络的结构非常简单，也正因为简单，所以可以使得输入到输出之间的路径的可解释性变得好一些。

除了这个例子，我还对 LunarLander 小游戏（就是那个飞船降落月球的小游戏）进行了实验。不过，这种直接迁移是有问题的，那就是：由于适应度函数本身的设置问题，导致飞船的降落过程虽然看上去很平稳，但是在接近地面的一瞬间，“一脚油”稳稳上天了……我只能说，搞科研来不得“刻舟求剑”。

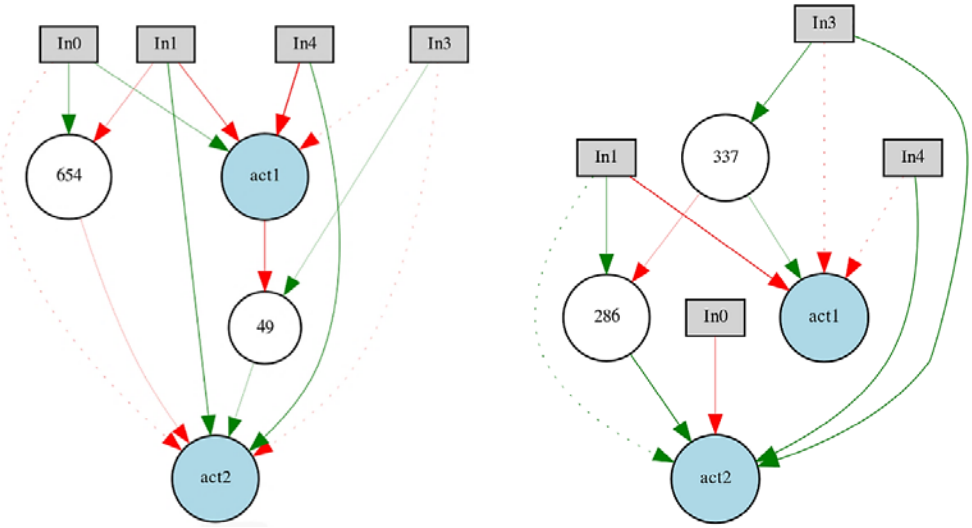


图 12-11 用 NEAT 玩 Cartpole 小游戏得到的网络拓扑图

12.4 本章小结

在这一章中，我们对 NEAT 算法的应用进行了初步的了解。可以说，NEAT 是一种相对“偏门”的训练策略机器人的方法。严格地说，NEAT 不是强化学习的研究范畴，建模的方式也和强化学习不太一样。从历史渊源看，NEAT 不仅没有完整的强化学习体系那么精美和健壮，也没有强化学习中那么多的子领域和研究方法。

以上这些，既是 NEAT 的缺点，也是 NEAT 的优点。

说是缺点，是因为 NEAT 更像一个孤立的解决问题的方法，而不是一个整体的、可沿着一条完整路径做改进的方案。在这个方面，基于强化学习和深度学习的方案就不一样了，因为在这个体系中，可以通过贝尔曼方程优化、环境试探的充分度、状态的估值是否高估、估值函数收敛等方面，判断深度学习网络是否已经出现过拟合，是否可以采用增加 L1/L2 惩罚项、调整学习率等手段来逐步优化。此外，别看 NEAT 新颖、有趣，但没有任何证据表明它的训练收敛速度一定比 DQN 或者 DDPG 好。

说是优点，是因为 NEAT 的理论体系还很单薄，所以也可以认为其中有太多尚未开垦的“处女地”。只要我们亲于钻研，结合相关的场景持续进行研究，也相对容易获得新的成果。

NEAT 就介绍到这里。我们还有很长的路要走，请别停下。

第 13 章　SerpentAI

不知不觉，本书已经写到第 13 章了。在前面的章节中，我们讲了不少算法模型，也介绍了著名的由 OpenAI 提供的 Gym 环境。如果你把前面的实验都做完，相信已经有了不少收获。不过，前面的实验有些美中不足——都是由 Gym 给我们提供完整的接口。

使用 Gym 接口的好处非常明显，即不需要以一帧完整的图像作为 State。一帧图像中少说也有数万像素，按照三通道图像来计算，就是 $200 \times 200 \times 3$——120000 维。而我们在 Gym 中使用的数据，通常都将维度降到了极其精简的程度。以第 12 章中的 Cartpole 和 LunarLander 两个实验为例，Cartpole 的输入状态只有 4 个维度，LunarLander 的输入状态也只有 8 个维度，与常见的图像相比，差别巨大。

这会直接导致如下两个后果。

- 很多算法在 Gym 环境中收敛迅速，很快就能得出结果。但是，在以屏幕像素为输入对象的情况下，收敛速度得不到保障。
- 如果我们想以其他环境作为实验对象，就显得非常困难了。在没有人给我们提供接口的情况下，需要我们自己实现接口，与各种软件或者游戏进行适配。先不说这件事情我们能不能做到，这件事情起码会让我们的工作重点发生比较大的偏移。

因此，如果我们有这种以其他软件或者游戏作为实验对象的实验场景需求，就需要代理工具来帮我们实现接口。在本章中，我们就来认识一个新的工具——SerpentAI。

13.1　简介

SerpentAI 是一个小有名气的开源软件，也是一个 Game Agent 框架。

SerpentAI 的 GitHub 地址为 https://github.com/SerpentAI/SerpentAI，其简体中文版的 Wiki 说明，可以访问 https://github.com/SerpentAI/SerpentAI/wiki/Home_zh_cn 获取。

SerpentAI 这个工具的功能一点也不简单，它有很多插件。为了把 SerpentAI 的运作原理用最简单的方式描述清楚，我画了一幅图，如图 13-1 所示。

一旦用 SerpentAI 启动了一个游戏，SerpentAI 就拥有了一系列功能。SerpentAI 会启动一个运行到“天荒地老”的循环，我们就叫它“死循环”吧。在这个循环的执行过程中，中间的循环体可以根据启动 SerpentAI 时设置的参数来设置其间隔，例如 2FPS、10FPS 等。如果是 2FPS，就表示每秒触发循环体 5 次；如果是 10FPS，就表示每秒触发循环体 10 次。一方面，这是根据业务的需要设置的，在一些游戏（例如棋牌类游戏）中，不需要把间隔设置得很短，可以设置为 2FPS 甚

至更久。另一方面，要看实验环境的硬件资源是否允许使用这么高的频率来刷新——别把 CPU 或者显卡“刷”得“着了火”。

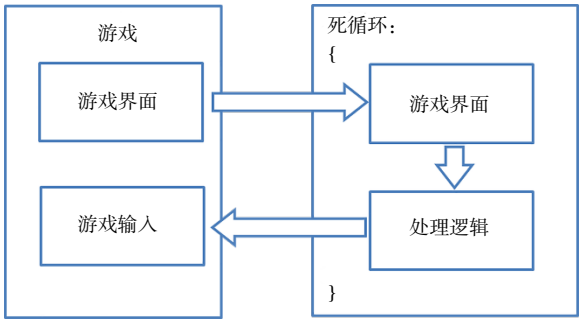


图 13-1　SperentAI 工作原理示意图

每次触发这个循环体之后，游戏的界面就会被输入循环体内。界面的分辨率，可以在启动 SerpenAI 的时候设置。在设置界面分辨率时，要注意平衡精细程度和单帧处理效率之间的矛盾，有效地进行取舍。

还是那句话——这帧图像输入之后，就是你说了算了。

你可以训练一个分类模型，以判断当前的页面是登录、退出、复活，还是其他任何状态。这种分类模型训练成功后，当一帧图像通过分类模型时，你就能够知道现在机器人所处的场景是什么，进而去做正确的动作了。例如，在登录场景中，就没必要让机器人做任何玩游戏的动作。为了把这个环节处理得高效一些，你可以自己做一个状态机来进行管理，简单地说，就是用一个枚举变量进行控制。进入游戏后，你就可以启动机器人来做任何事情了——不论是训练，还是用训练好的模型玩游戏。在这个环节，你可以对游戏程序做各种动作。这些动作，可以是鼠标操作（例如单击、拖拽等），也可以是按下键盘上的任何一个按键。

这就是模拟一个人玩游戏的操作场景了。当然，游戏程序并不知道玩家是人还是由人训练的机器人，它只会根据输入去做相应的动作，进而使屏幕上显示的内容发生变化。

接下来，就是再次完成这样一个过程——周而复始。

怎么样，原理不复杂吧？下面我们看一下具体怎么使用 SerpentAI。

13.2　安装和配置

SerpentAI 的安装过程比较简单，请读者参考附录 A.7.3 节。

SerpentAI 使用 pyenv 作为其虚拟环境的载体。与 Conda、Docker 等虚拟容器不同，pyenv 显得更“轻”，是 2018 年 SerpentAI 推荐的环境。在 2017 年，SerpentAI 还在以 Conda 为主要的虚拟容器载体呢——变化真的很快。

在安装 SerpentAI 的过程中，需要注意包的相互依赖及版本依赖——这是开源环境中永恒的

梦魇。此外，SerpentAI 可以安装自己的 ML 模块，就是一个 1.4.1 版本的 TensoFlow 环境。如果你安装了这个 ML 模块，将主要使用 tf.keras 搭建 Inception 来做场景识别。这是一种便于我们自动化地训练 Agent 的重要手段。例如，对一个游戏，如果希望在游戏结束后，Agent 能够自动单击“退出”“重新开始”这样的按钮来重启游戏，或者重启一个 Episode，而不是由人来进行操作，就需要通过一个小的分类器对页面进行分类处理，如图 13-2 所示。

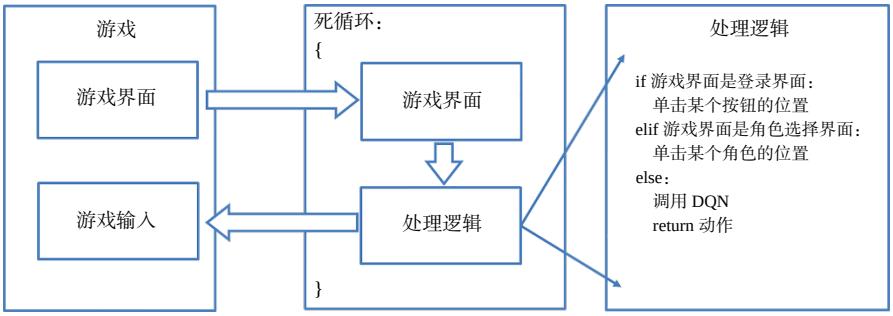


图 13-2　SperentAI 的自动训练流程

如果使用 PyTorch，可以继续使用 tf.keras 来做场景识别训练，以节省在搭建上消耗的时间。另外，可以用 PyTorch 来训练 Agent AI。在 13.3 节中，我们就通过几个例子来熟悉一下 SerpentAI 关键功能的使用。

为什么要把这样一个开源工具作为专门的一章来介绍呢？因为有了它之后，一切程序环境都能化身为供我们“玩耍”的试验场，而不局限于 Gym，从而在更大的范围内训练智能机器人。

13.3　示例

13.3.1　创建 Game Plugin

在这一节中，我们将通过一个“Hello World”程序了解一个完整的 Game Plugin 是如何建立起来并与游戏进行交互的。

首先，在 Ubuntu 环境下安装一个“老少咸宜”的游戏——《雷电》（文件包名为“chromium-bsu”），命令如下，如图 13-3 所示。

```
sudo apt-get install chromium-bsu
```

如果要启动这个游戏，直接使用 Shell 命令就可以了，具体如下。

```
chromium-bsu
```

然后，建立 Game Plugin，命令如下，如图 13-4 所示。

```
serpent generate game
```

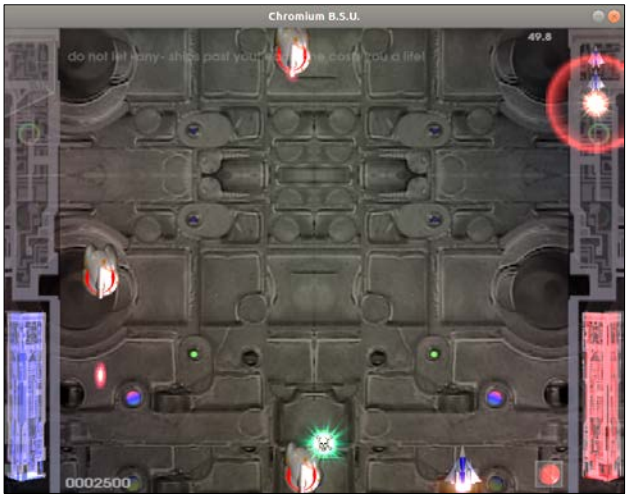


图 13-3 《雷电》游戏

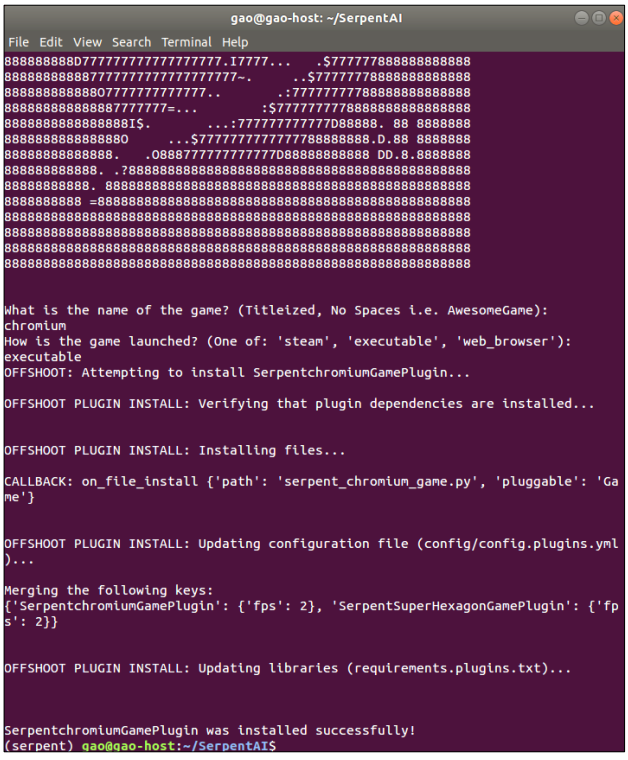


图 13-4 建立 Game Plugin

在这个过程中，程序会问我们“这个游戏叫什么名字”，此时我们输入即将用来做实验的游戏

的名字即可（也可以随便起一个名字，使用即将用来做实验的游戏的名字主要是为了记忆方便）。在这里，输入“chromium”，也就是“雷电”的英文名称。

随后，程序会问我们“游戏的启动方式是什么”。有 steam、executable、web_browser 三个可选项。“steam”就是指著名的 steam 平台，即用 steam 平台启动的游戏；“executable”就是指各种桌面 App，这应该是使用最多的；“web_browser”就是指网页游戏，也就是用网页启动一个嵌入 HTML5 页面的 Flash 或者 JavaScript 形式的游戏。

很快，这个叫作“SerpentchromiumGamePlugin”的工具就算建立完成了。在这个过程中会弹出一些信息，内容蛮多的，比较有用的是中间的“’fps’: 2”，表示 SerpentAI 以每秒 2 帧的速度和游戏进行交互。

《雷电》这个游戏，怎么说呢——简单、省“脑子”，看上去是在打敌人，实际上是在躲子弹。如果你玩过《沙罗曼蛇》，就可以把《雷电》当成竖屏版《沙罗曼蛇》。你可以试着玩玩《雷电》，它的套路非常简单，玩得好不好主要取决于玩家的反应速度。

建立 Game Plugin 后，就可以进入 Game Plugin 所在的目录来查看它的结构了。如图 13-5 所示，在我使用的环境中，Game Plugin 安装在 ~/SerpentAI/plugins/SerpentchromiumGamePlugin 目录下。

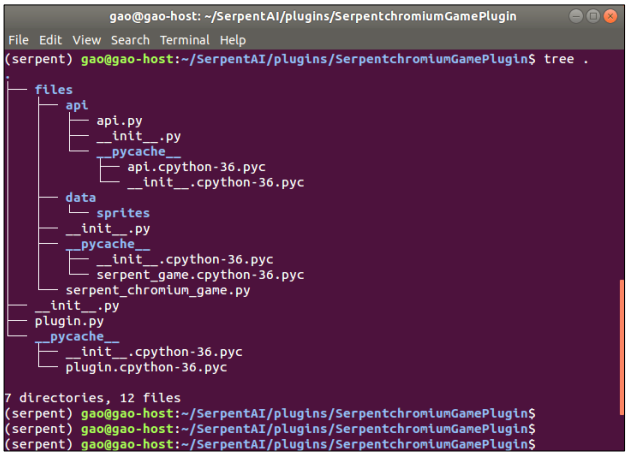


图 13-5　Game Plugin 目录结构

这个“~”就是“home”位置，也就是我的实验环境中所对应的“/home/gao/”的位置。编辑文件 serpent_SuperHexagon_game.py，命令如下。

```
vim ~/SerpentAI/plugins/SerpentchromiumGamePlugin /files/serpent_chromium_game.py
```

在该文件中，将“WINDOWS_NAME”改成“Chromium B.S.U.”，将“EXECUTABLE_PATH”改成“/usr/games/chromium-bsu”。

来到 ~/SerpentAI 目录下，启动游戏，如图 13-6 所示。请注意，在此之前，务必将 steam 安装在你的 Ubuntu 环境中，不然会报错的。

```
cd ~/SerpentAI
serpent launch chromium
```

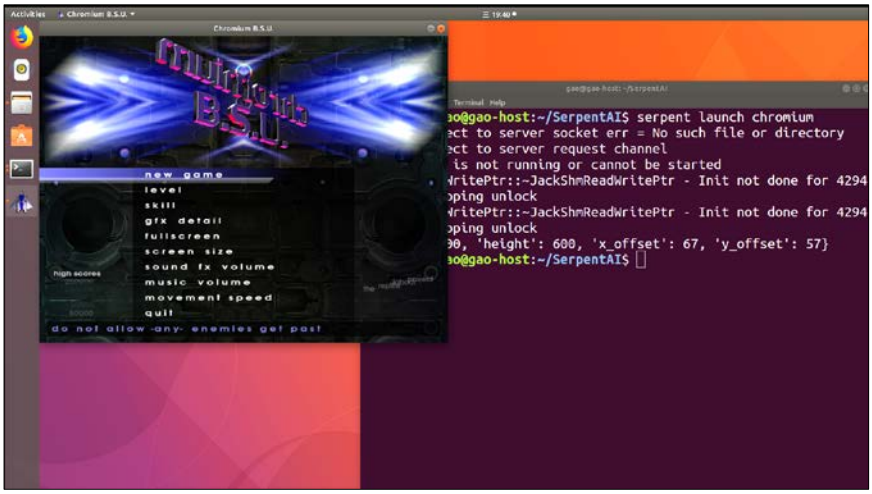


图 13-6　通过 Game Plugin 正常启动游戏

然后，游戏会被 SerpentAI 启动，出现在桌面的左上角（就像图 13-6 中那样，表示正常启动了）。如果你拿不准“WINDOWS_NAME”应该叫什么名字，那就使用

```
serpent window_name
```

命令来获取焦点。此时，SerpentAI 会告诉你它的名字——就是窗口顶端的标题。

13.3.2　创建 Game Agent

创建 Game Plugin 后，就可以创建 Game Agent 了。

Games Plugin 相当于 SerpentAI 建立的一个针对游戏的接口，可以模拟与游戏的交互、读取一帧图像、输出一个动作。Game Agent 则真正承担了这部分处理逻辑的内容。

执行如下命令。

```
serpent generate game_agent
```

这时，SerpentAI 会让你给 Game Agent 起一个名字。我们输入如下内容。

```
chromium
```

还是《雷电》这个游戏，因此我们还是使用“chromium”这个名字，如图 13-7 所示。

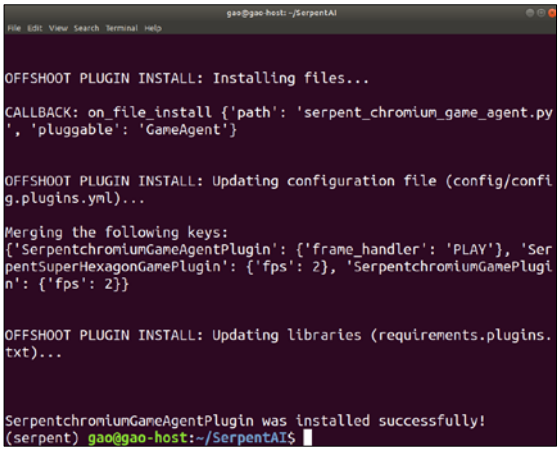

gao@gao-host: ~/SerpentAI
File Edit View Search Terminal Help
OFFSHOOT PLUGIN INSTALL: Installing files...
CALLBACK: on_file_install {'path': 'serpent_chromium_game_agent.py', 'pluggable': 'GameAgent'}
OFFSHOOT PLUGIN INSTALL: Updating configuration file (config/config.plugins.yml)...
Merging the following keys:
{'SerpentchromiumGameAgentPlugin': {'frame_handler': 'PLAY'}, 'SerpentSuperHexagonGamePlugin': {'fps': 2}, 'SerpentchromiumGamePlugin': {'fps': 2}}
OFFSHOOT PLUGIN INSTALL: Updating libraries (requirements.plugins.txt)...
SerpentchromiumGameAgentPlugin was installed successfully!
(serpent) gao@gao-host:~/SerpentAI$

图 13-7　成功建立 Game Agent

别急，还没完。现在目录下面已经多了一些程序。文件 serpent_chromium_game_agent.py 位于 ~/SerpentAI/plugin/SerpentchromiumGameAgentPlugin 目录下，如图 13-8 所示。

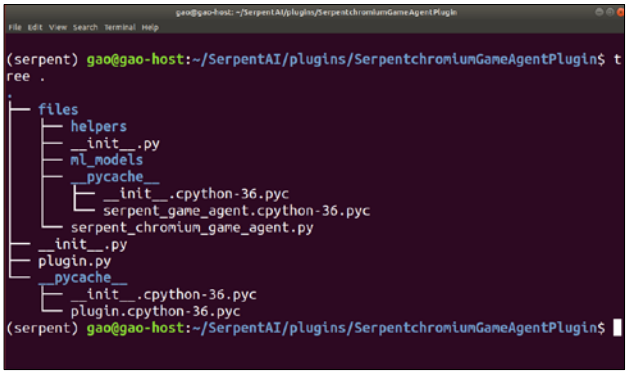


图 13-8　Game Agent 路径结构

打开这个文件，并将代码添加到对应的方法的位置，具体如下。

```
from serpent.input_controller import KeyboardKey

…

def handle_play(self, game_frame):
      self.input_controller.tap_key(
KeyboardKey.KEY_SPACE)
      print('hello')
```

这时就可以使用 Game Agent，通过 Game Plugin 来玩游戏了。

在启动之前要注意一个细节：可能需要重启 Ubuntu 17.10，从而以 Ubuntu on Xorg 的方式引导桌面。

Ubuntu 17.10 默认是以 Wayland 方式来引导桌面的，而这会导致当前版本的 SerpentAI 无法正常工作，所以，需要用 Ubuntu on Xorg 的方式来引导它。引导成功后，进入桌面，启动 Terminal，如图 13-9 所示。

图 13-9　以 Ubuntu on Xorg 的方式引导桌面

连续执行如下命令。

```
serpent launch chromium

serpent play chromium SerpentchromiumGameAgent
```

你会发现，这个 Agent 已经可以正常工作了。具体看到的现象就是，在 1 秒内小飞机会开枪两次。这是

```
self.input_controller.tap_key(KeyboardKey.KEY_SPACE)
```

的作用。此外，在 Terminal 里面会不断显示“hello”，这是 `print("hello")`的作用。

现在，Agent 已经能够正常地和一个游戏进行交互了，如图 13-10 所示。因此，我们可以开始训练这个游戏机器人了。

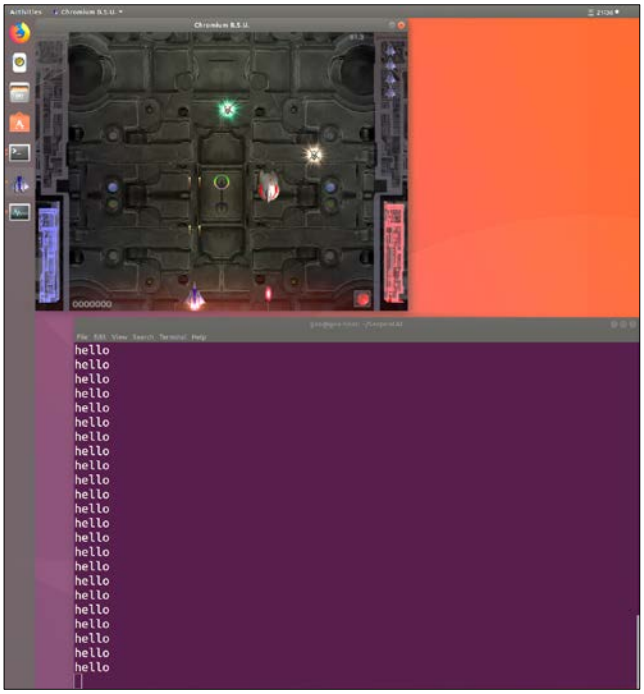


图 13-10　Agent 接管游戏

13.3.3　训练 Context Classifier

1. 规则分析

“Context Classifier”翻译成中文应该是“上下文分类器”。这个名字非常奇怪，我们根本无法从字面意思来理解它是做什么用的。

准确地说，Context Classifier 就是前面提到过的那个可以对页面进行分类处理，以判断究竟需要单击哪个按钮才能让游戏自动重新登录、选择难度等的小分类器。因此，我们还是要先自己玩一玩游戏，了解游戏这个状态机是怎么运作的。

就拿《雷电》这个游戏来说吧，在游戏的起始界面选择“开始”就可以了，即直接按“空格”键。进入游戏后，“上”“左”“下”“右”“开枪”“放保险”……只有几个可以做的动作。

“上”“左”“下”“右”这四个方向键就不用说了，它们的功能就是纯粹的方向移动，且游戏中似乎没有惯性现象。“开枪”需要不停地按下按钮才能做到。“放保险”这个说法是玩过“街机”的“70 后”“80 后”才喜欢用的词汇，同义词就是“放大招”，即使用后全屏的敌人都“死翘翘”的一种技能。不过，在这个游戏里面，“放保险”的功能很奇怪，那就是按两下“回车”键后会产生一种“殉爆”的效果——自己爆炸，损失一条“命”，并发出一个向四面扩散的巨大的冲击波，碰到哪个敌人，哪个敌人就“挂”了。

此外，如果被敌人击中或者发生碰撞，就会损失“血量”。随着“血量”的减少，最后飞机会爆炸，玩家损失一条“命”。

如果有敌人的飞机飞到屏幕最下方并飞出屏幕，就会在屏幕下方发出一片红光，这同样会对玩家的飞机造成伤害，准确地说是直接损失一条“命”。

在屏幕中从上到下飞行的，不仅有敌人的各种奇形怪状的飞机和子弹，还有一些奇形怪状的武器增强工具——就是我们俗称的“枪”，“吃”了之后火力就会增强。

如果所有的“命”都用光了，屏幕中间会有大大的“loser”字样出现。然后，按任意键，游戏就会重启。

游戏界面左下角显示的是得分。据我观察，不仅击杀敌人的飞机能得分，“吃”掉“枪”也能得分，暂时没有发现扣分的情况。

游戏的内容大概就这么多，而且是无限循环的，界面如图 13-11 所示。

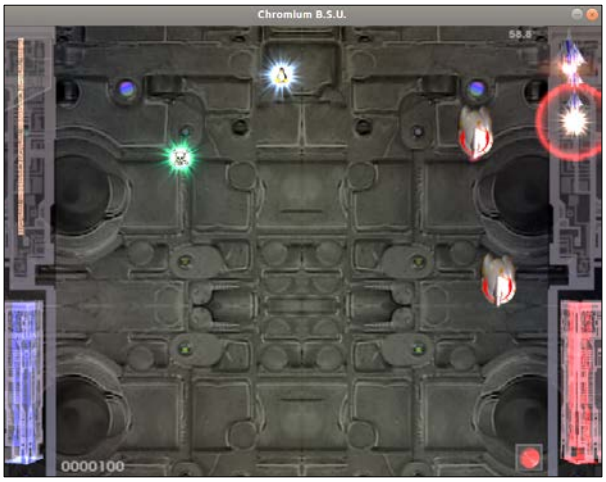


图 13-11 《雷电》游戏界面

《雷电》游戏的难点有如下两个。

- 难点一：和其他使用 SerpentAI 作为代理工具的游戏一样，没有第二套接口来获得环境数据，所有的信息都是从图形界面上得到的。这就意味着，如果要降维，就必须自己开发相应的处理工具对输入的图形界面进行处理。例如，截取左下角部分量化“血量”、截取左上角部分量化“枪”的强度、截取右上角部分量化剩余几条“命”等。否则，能帮助机器人学到有用信息的数据将被淹没在一幅图片中，我们也只能采用其他办法来训练了。
- 难点二：交互的速度。我们建立的这个插件的交互速度目前是 2FPS，即每秒交互 2 次。这样的速度够吗？如果按照这个速度，应该是 120APM[①]的“手速”——绝对不算慢了。“手

① Action Per Minute，每分钟的操作次数，是一个用于描述游戏玩家操作速度的重要指标。

速”是在即时战略类游戏里面才会使用的词，在打飞机类的游戏中，从来没有听过这种说法。每秒按键 2 次够吗？我觉得不够。在一秒内按键 2 次，连一架飞机都打不下来——这些飞机看上去都超级“耐揍”啊。因此，需要将 FPS 的值加大，例如修改成 8。这一点可以通过修改文件迅速解决，具体如下。

```
vim ~/SerpentAI/config/config.plugin.yml
```

可以被修改为

```
SerpentchromiumGameAgentPlugin:
  frame_handler: PLAY
SerpentchromiumGamePlugin:
  fps: 8
```

如果你觉得自己的机器性能非常好，将 FPS 的值改成 8 也不过瘾，再增大一些也无妨。

2. 场景分析

按照这样的规则来分析，其实在游戏的开始页面是不用进行判断的（因为我们只会经过开始页面一次），只要手动按一下“空格”键就可以进入游戏了。

一旦进入游戏，应该说，就可以“无差别”地玩游戏了。不管是用 MCTS 还是 TD，都可以这么考虑。动作无非就是“上”“左”“下”“右”“开枪”“放保险”。好吧，现在似乎看不出要不要区分这个状态和另一个“输掉”的状态。

所谓“输掉”的状态，就是屏幕中间出现大大的“loser”字样，按任意键开始新一轮游戏的情况。可以肯定的是，即便不识别这个状态，按任意键也能让游戏重新开始（游戏不会停在这里过不去），所以要不要识别它就无所谓了。但是，在有些游戏中是不行的——游戏结束时，会在屏幕上弹出一个菜单，让玩家单击类似“NEW GAME”的按钮。如果不能识别这样的场景，就自然不会去单击相应位置的按钮，也就不能开始新游戏了——你肯定不愿意看机器人玩一宿游戏吧。

从游戏目前的设定来看：如果策略不好，那么存活的时间一定是非常短的；如果策略好，就一定能——闪转腾挪躲子弹，积极备战吃枪杆，利利索索打敌人，健健康康走得远……

因此，可以考虑识别“Game”状态和“Lose”状态，并以此作为标杆。不为别的，只为评价当前这个 Episode 究竟还有多长时间结束。

下面我们就训练一个这样的分类器，用于识别“Game”状态和“Lose”状态（分别表示正在玩游戏和游戏失败）。

3. 获取样本

获取样本，就是利用类似录屏的功能录下若干“Game”状态的游戏界面，以及若干“Lose”状态的游戏界面，然后通过训练一个分类器来实现二分类。

样本的获取，我觉得要注意如下两点。

- 样本的丰富性问题。每个场景的样本要尽可能丰富。例如，在本节的示例游戏里面，要想使样本更丰富，就要使“Game”状态和“Lose”状态都有尽可能多的不同的图片。
- 样本要均衡。这是所有分类问题离线计算都需要考虑的。例如，对“Game”和“Lose”这两个类别，要么都训练 100 幅，要么都训练 200 幅，不要一个多、一个少。歪斜的分类器对样本数量较少的类别的识别效果很差。

现在，开始录制样本。

先录制“Game”类，也就是处于游戏状态的类别。启动游戏，命令如下。

```
serpent launch chromium
```

然后，输入如下命令。

```
serpent capture context chromium 0.5 game
```

其中，`0.5` 是指每 0.5 秒截屏一次，`game` 是指类别名称。

你可以在游戏界面上尽情地玩，同时，在 Terminal 里面会显示现在截屏的帧数。在 Terminal 里面按“Ctrl+C”快捷键终止游戏，SerpentAI 会把截屏信息保存到 ~/SerpentAI/datasets/collect_frames_for_context/game/目录下。

在这个目录下，都是以“frame”开头、以“.png”结尾的图片文件。SerpentAI 官方建议，一个类别下面至少要有 200 幅图片。我截取了 300 多幅，然后删掉了那些不小心录上“loser”字样的，最终保留了 300 幅。如法炮制，录制“Lose”状态的屏幕，也保留 300 幅图片。

“Game”样本和“Lose”样本，如图 13-12 所示。

图 13-12 “Game”样本和“Lose”样本

从录制样本的情况看，用于区别两者的特征只是屏幕中间的“loser”字样，但是，如图 13-12 所示，右边图片中的一架敌方飞机朝玩家“葬身”的地方不停地发射激光，这个特征不仅非常明显，也和左边图片呈现了巨大的反差。因此，我们必须人为降低这类图片的比例——可以有，但不能让它们过多地出现在样本中，否则将非常容易导致过拟合。

4. 训练

SerpentAI 的训练过程和 MNIST 几乎一样。所以，对图形分类这种典型的离线任务来说，有 GPU 会更好。如果有 GPU，可能几分钟就能完成这个任务了。但如果只有 CPU……起码可以去吃顿饭再回来看结果。

输入如下命令。

```
serpent train context 15
```

这个 15，就是指 15 个 Epoch。

在这里必须加入一个小插曲，那就是：1.4.1 版的 TensorFlow 在训练的过程中会报错，显示“softmax() got an unexpected keyword argument 'axis'”字样。没办法，开源软件版本之间的协调就是很麻烦，没地儿说理去。因此，我只好再去安装 TensorFlow 1.7.1，命令如下。

```
pip3 uninstall tensorflow

pip3 install tensorflow==1.7.1
```

然后，SerpentAI 会自动下载 Inception V3，启动 TensorFlow 进行训练，如图 13-13 所示。这个过程是自动完成的，很省心。

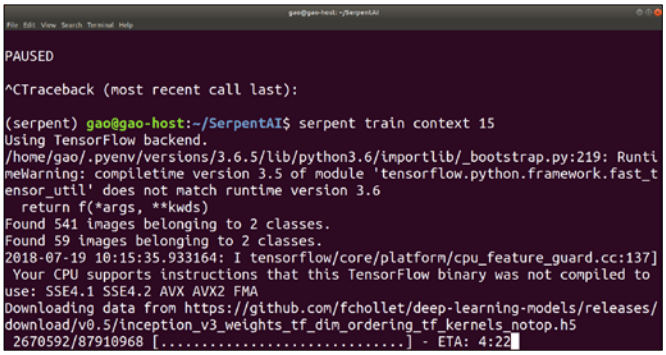


图 13-13　下载 Inception 并进行训练

在这个过程中，SerpentAI 会自动把我们刚刚收集的数据按照 9 比 1 的比例分为 Training 和 Validation 两个数据集，分别放到 ~/SerpentAI/datasets/current 中去。

在做这个实验时，我使用的是 Ubuntu 17.10 的虚拟机，4 核 Intel i7 7700HQ，2.8GHz，睿频 3.4GHz——尽管能够运行，但 8GB 内存，1TB SSD 硬盘（如图 13-14 所示），全程风扇嗡嗡作响，像直升机要起飞了似的，结果真的花了半天时间才训练好……因此，买显卡是很有必要的，否则，问题很严重……

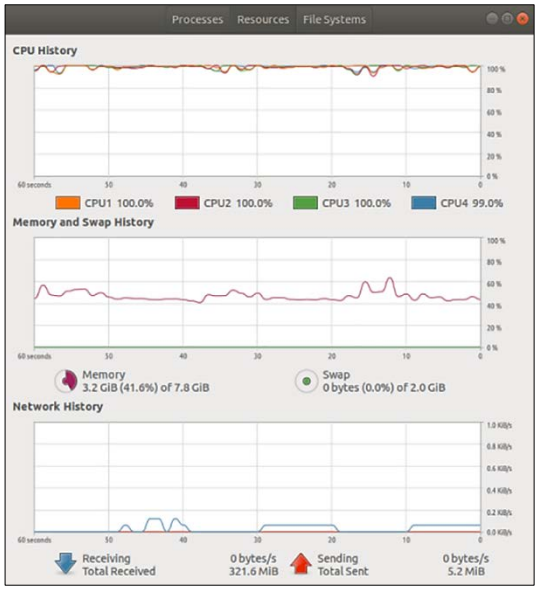


图 13-14　CPU 和内存占用

5. 实测

训练结束后，模型会保存在 ~/SerpentAI/datasets/context_classifier.model 目录下。

如图 13-15 所示，训练效果有点惨不忍睹。这种输出格式就是 Keras 输出的典型格式，我们就以最后一个 Epoch 为例说明一下吧——15/15。

- “acc: 0.9297”表示在训练集上的准确率为 92.97%。
- “val_acc: 0.6265”表示在验证集上的准确率为 62.65%。

```
Epoch 8/15
16/16 [==============================] - 1299s 81s/step - loss: 0.3390 - acc: 0.8710 - val_loss: 0.5074 - val_acc: 0.8130
Epoch 9/15
16/16 [==============================] - 1300s 81s/step - loss: 0.2471 - acc: 0.9151 - val_loss: 0.6364 - val_acc: 0.6954
Epoch 10/15
16/16 [==============================] - 1366s 85s/step - loss: 0.1801 - acc: 0.9254 - val_loss: 1.7096 - val_acc: 0.4406
Epoch 11/15
16/16 [==============================] - 1380s 86s/step - loss: 0.1521 - acc: 0.9277 - val_loss: 0.5347 - val_acc: 0.6776
Epoch 12/15
16/16 [==============================] - 1381s 86s/step - loss: 0.3607 - acc: 0.8678 - val_loss: 0.7965 - val_acc: 0.5600
Epoch 13/15
16/16 [==============================] - 1353s 85s/step - loss: 0.0968 - acc: 0.9649 - val_loss: 0.6932 - val_acc: 0.6259
Epoch 14/15
16/16 [==============================] - 1348s 84s/step - loss: 0.1951 - acc: 0.9277 - val_loss: 0.4142 - val_acc: 0.8646
Epoch 15/15
16/16 [==============================] - 1423s 89s/step - loss: 0.2108 - acc: 0.9297 - val_loss: 0.7115 - val_acc: 0.6265
Success! Model was saved to 'datasets/context_classifier.model'
(serpent) gao@gao-host:~/SerpentAI$
```

图 13-15　Context Classifier 的训练结果

过拟合现象非常严重……半天时间算是白费了，需要重新训练。其中，“loss”和“val_loss”分别指在训练集和测试集上的损失函数。现在，我们可以不看这两个指标（由于没有量纲，这两

个指标的参考意义不大），只看准确率就够了。

不过，这么低的准确率会严重影响 Agent 玩游戏的过程，因此需要对其进行改进。

我们先把流程完成，再做改进。让 Agent 来处理每一帧图像，并让它们通过网络正向传播，产生分类。把训练好的模型复制到指定位置，命令如下。

```
cp ~/SerpentAI/datasets/context_classifier.model
~/SerpentAI/plugins/SerpentchromiumGameAgentPlugin/files/ml_models
```

修改程序，具体如下。

```
~/SerpentAI/plugins/SerpentchromiumGameAgentPlugin/files/serpent_chromium_game_
agent.py
```

把下面两个段落添加到程序中。

```
def setup_play(self):
       import offshoot
       plugin_path = offshoot.config["file_paths"]["plugins"]

       context_classifier_path = f"{plugin_path}/SerpentchromiumGameAgentPlugin/
files/ml_models/context_classifier.model"
       from serpent.machine_learning.context_classification.context_classifiers.cnn
_inception_v3_context_classifier import CNNInceptionV3ContextClassifier
       context_classifier = CNNInceptionV3ContextClassifier(input_shape=(384, 512,
3))  # Replace with the shape (rows, cols, channels) of your captured context frames
       context_classifier.prepare_generators()
       context_classifier.load_classifier(context_classifier_path)
       self.machine_learning_models["context_classifier"] = context_classifier
```

```
def handle_play(self, game_frame):
       self.input_controller.tap_key(KeyboardKey.KEY_SPACE)
       context = self.machine_learning_models["context_classifier"].predict(game_
frame.frame)
       print(context)
```

上面的段落表示在 Agent 启动的时候一次性调用模型。下面的段落表示在处理每一帧时需要按“空格”键，以及预测当前游戏界面的帧 `game_frame.frame` 是什么类型的并把类型打印出来，如图 13-16 所示。

误判挺严重的，明明还处在“Game”状态，却告诉我们“Lose”了。要想解决这个问题，可以从如下两个方面考虑。

- 添加样本，提高样本的丰富程度。也就是说，“Game”状态和“Lose”状态的样本数量都要增加。不过，再多的样本量也只能提高准确率，不可能做到百分之百准确。所以，还要加入一个技巧——“平滑”。
- 所谓“平滑”，就是根据连续多个帧的预测信息取平均值来确定分类。例如，在连续的 5 帧

图像里面，如果有 3 帧或者 4 帧都处在“Game”状态，就说明游戏正在进行，如果有 3 帧或者 4 帧都处在“Lose”状态，就说明游戏结束了。可以用这样的方法抹平由误判产生的信号毛刺。

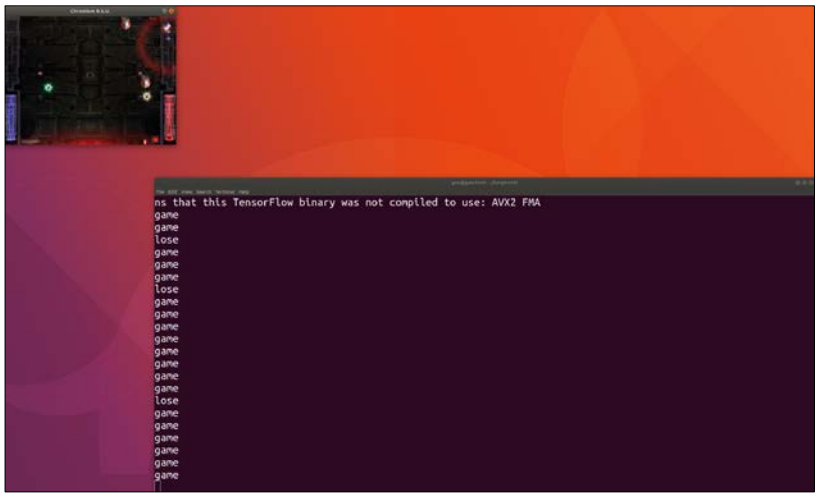


图 13-16　Context Classifier 的工作状况

到这里，有关 Context Classifier 的训练和使用就作为一个独立的部分介绍完了。关于其识别的正确率，可以和 Agent 策略的好坏分开讨论，以两个不同工程的视角分别建模来解决。

对二分类问题来说或许不明显，分类一旦增多，Inception V3 网络的复杂度势必会提高——这是 SerpentAI 自动完成的。网络越复杂，正常传播所需要的时间就越长，也就意味着，每一次使用 `handle_play` 这个方法来处理逻辑所消耗的时间越长。其后果可能是灾难性的。例如，在一些对输入效率要求比较高的游戏中，可能干脆没办法训练了（因为每帧的处理时长可能已经达到了数百毫秒，8FPS 恐怕不够用了）。

6. Region 对象

在 SerpentAI 中，还有一种方式可以有效地改善这种分类误差，那就是对 Region 进行分类。

提出 Region 这个概念的目的是细化和丰富对屏幕信息的处理方法，并非只针对改进 Context Classifier 这一项技术。Region 是指在页面的某个位置定义一个矩形，给这个矩形取一个别名，然后对这个别名所指代的矩形的位置进行一系列操作。例如，可以对 Context Classifier 方法进行改进，变成针对一个矩形的方法，从而节省时间、提高正确率。尺寸变小，特征就更容易提取——这是一种人为的降维。在一次对分类器的正向传播中，也可以通过这种方法来节省时间。

下面我们就演示一下如何实施这种改进。

（1）定义 Region

定义 Region，就是在屏幕上定义一个矩形，也就是声明一个矩形的位置。

为了预测要定义的位置，SerpentAI 官方推荐使用 Spritex 工具进行测量，如图 13-17 所示。定义的位置信息，是一个由 4 个值表示的矩形。这 4 个值分别是：矩形左上角的 y 和 x，以及矩形右下角的 y 和 x。这 4 个值足以确定一个矩形的位置了。

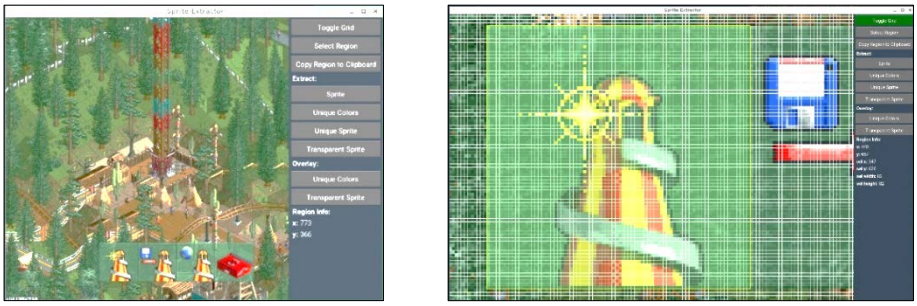


图 13-17　用 Spritex 测量位置（1）

启动 Spritex，测量某个位置。例如，我们想要用 Spritex 来测量屏幕上“loser”字样的位置，可以打开任何一个包含“loser”字样的截屏文件，具体如下，如图 13-18 所示。

```
spritex datasets/collect_frames_for_context/lose/frame_1531965836.4383726.png
```

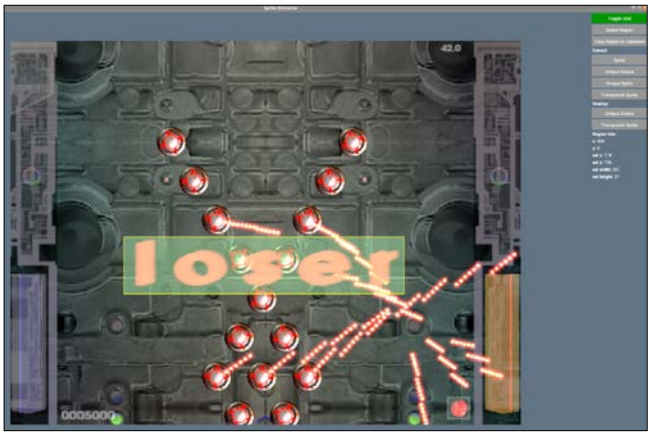


图 13-18　用 Spritex 测量位置（2）

测量结果为：sel x，114；sel y，196；sel width，281；sel height，57。那么，Region 的描述就应该是

$$(196,114,253,395).$$

不过，请注意，一会儿要训练的 Inception V3 模型的最小输入尺寸为 139×139，所以 57×281 的大小是不合格的。对于其高度，我们可以在纵向移动 41 像素，使其变成

$$(155,114,294,395).$$

如果你觉得安装 Spritex 麻烦，或者觉得用不惯，可以试试 QQ 的“Ctrl+Alt+A”功能——未必做不了这件事。

拿到 Region 的信息以后，就需要对它进行定义。

打开文件 ~/SerpentAI/plugins/SerpentchromiumGamePlugin/files/serpent_chromium_game.py，修改对应位置的内容，具体如下。

```
    @property
    def screen_regions(self):
        regions = {
            "STATUS": (196, 114, 477, 171)
        }
```

（2）截屏

启动游戏，命令如下。

```
serpent launch chromium
```

启动截屏功能，命令如下。

```
serpent capture region chromium 0.5 STATUS
```

大概玩 2 分钟左右——其中 1 分钟是正常玩游戏，1 分钟是游戏结束后显示“loser”字样。这些截屏信息都保存在 ~/SerpentAI/datasets/collect_frames/STATUS 目录下。

现在我们已经有两种不同的屏幕图片了，可以考虑把它们分成两个不同的组——分别复制普通的“Game”状态和“Lose”状态的截屏文件，放到对应的文件夹里面去，如图 13-19 所示。

目录分别如下。

- ~/SerpentAI/datasets/collect_frames_for_context/game
- ~/SerpentAI/datasets/collect_frames_for_context/lose

检查一下，看看有没有复制错的，然后对图片进行挑选和删除。

删除原则如下。

- 删除内容重复的图片，例如两幅完全一样或者几乎一样的图片。
- 样本数量要平衡。在这里，两个图片类别都处理成了 120 个样本，剩余的都被我删除了。

图 13-19　截屏信息

（3）训练

把数据放到文件夹里，启动训练，命令如下。

```
cd ~/SerpentAI

serpent train context 8
```

这一次的训练速度快了很多。如图 13-20 所示，训练一个 Epoch 大约需要 1 分钟，训练 8 个 Epoch 也就是 8 分钟左右，这得益于减少了样本数量和减小了样本尺寸——立竿见影。这些样本的表现也不错，训练集的正确率为 95.71%，测试集的正确率居然达到了 100%。训练集在这 8 个 Epoch 上的表现出现了比较严重的“颠簸”现象（不是稳步上升的），这主要是整个样本集中的样本数量偏少所致。为了改进这个问题，需要增大训练样本集和验证样本集。

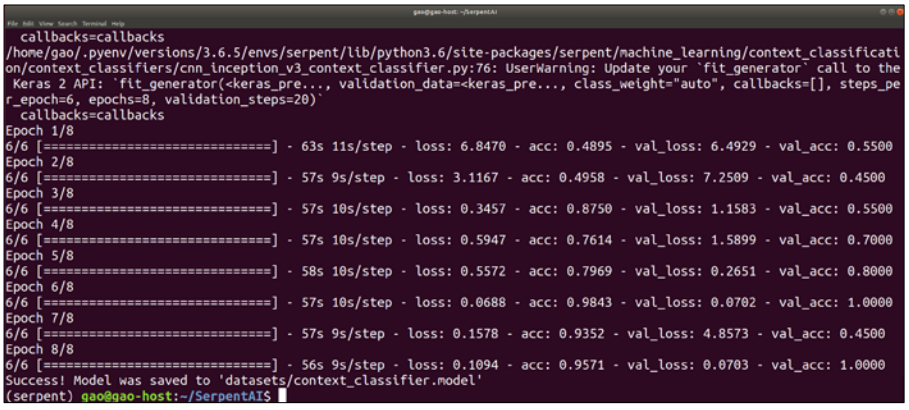


图 13-20　训练结果

（4）分类测试

最后，进行分类测试。

修改文件，命令如下。

```
vim plugins/SerpentchromiumGameAgentPlugin/files/serpent_chromium_game_agent.py
```

有三个位置需要添加内容，具体如下。

```
import cv2
…
    def setup_play(self):
     context_classifier = CNNInceptionV3ContextClassifier(input_shape=(139, 281, 3))
#修改模型输入的图片尺寸

    def handle_play(self, game_frame):
             self.input_controller.tap_key(KeyboardKey.KEY_SPACE)
            slogan = game_frame.frame[155:294, 114:395]
#截取相应的 Region 位置
            context = self.machine_learning_models["context_classifier"].predict
(slogan)
            print(context)
```

还要把新的模型赋值到对应的位置，具体如下。

```
cd ~/SerpentAI

cp datasets/context_classifier.model plugins/SerpentchromiumGameAgentPlugin/files/
ml_models/
```

进行测试，命令如下。

```
serpent launch chromium

serpent play chromium SerpentchromiumGameAgent
```

但是，这样训练的效果仍旧不太好，原因很简单——分类器中的样本数量太少。后来，我增加了样本数量，在每个类别接近 500 个样本的时候，正确率才相对比较稳定。根据我的测试，当两个分类的样本数量各自在 420 个左右的时候，训练 8 轮，在训练集中的正确率大约为 96%，在验证集中的正确率也在 96% 左右。启动游戏以后，测试结果是一致的。这样的分类器基本就可以考虑使用了。接下来，使用一下平滑技巧就可以了。

13.3.4 训练 Agent

1. 奖励值

到 13.3.3 节为止，从游戏屏幕上获取一帧图像，截取部分像素，训练和使用分类器模型，向

游戏输出一个动作——一个得到完整 Transition 的过程就完成了。除了没办法显式地获得奖励值以外，一切都跟 Gym 环境很接近了。

有关奖励值的获得，可以考虑从一帧屏幕图像的某个位置截取一个区域，然后让它通过 OCR 模块翻译成数字或者文字。此外，好像没有更好的办法可以直接获取奖励值了。虽然 SerpentAI 官方一直说 OCR 模块很快就可以释放出来，但在写作本书时，SerpentAI 还没有提供 OCR 模块。

在 SerpentAI 的安装过程中设置 OCR 模块，就相当于安装 tesserocr 模块。

tesserocr 模块是一个提供了 Python 接口的 OCR 开源项目，使用起来很简单。我们只能截取屏幕上的一个小范围区域，然后通过 tesserocr 模块得到一个数字（例如一个分数），用当前 Step 的得分减去上一轮的得分，将这个差值作为奖励值。这个思路是可以尝试的。

当然，也可以不这样做。使用蒙特卡罗法求估值也是可以的。例如，只要一个 Step“存活”下来，就给 1 分的奖励，然后用贝尔曼方程慢慢求出各个状态的 $Q(s,a)$。不过，这个想法似乎跟 DDPG 的思路更为相似。

如果每个 Step 的奖励值都是 0，直到最后“loser”字样出现时给一个 –1 的奖励值来表示惩罚（游戏结束），就是蒙特卡罗法了。

2. 模型

到了这一步，就可以考虑开始训练一个带有策略的 Agent 了。你想用什么来设计模型呢？是 DQN 还是 DDPG，抑或它们各自的 A3C 版本？该怎么设计这个模型呢？

我觉得，使用 DQN 或者 DDPG 都是可以的。在这里，为了演示 SerpentAI 的使用，我们就使用 DQN 来设计一个包含 Main Network 和 Target Network 的网络并完成训练。

理论上，如果计算资源足够，并可以支持“多进程+多协程”环境，首先应该想到用 A3C 的方法来训练。只不过，要想训练这样一些 Agent，需要开启多个游戏副本，每个 Agent 在交互的过程中都要获取对应的游戏窗口的焦点——这就比较麻烦了。

虽然焦点切换很容易实现（用程序就能控制），但这个切换过程需要时间。也就是说，象棋、围棋一类的回合制游戏对于延迟是不敏感的，而《雷电》这种游戏则不然，哪怕只有 0.2 秒的延迟，都会产生很大的影响。所以，在单机上训练 A3C 版本的《雷电》游戏机器人是不可行的。

如果真的要这样做，我能想到的最靠谱的办法是：找几台计算机，将每一台作为一个 Agent，在其本地玩《雷电》游戏，然后共享一个 Target 网络（这些计算机需要通过比较可靠的网络连接在一起）。在这里不推荐使用家用路由。建议使用企业级路由，以进一步降低延迟。

总之，一定要注意网络的延迟问题。对 CPU 这种运算速度极快的元件来说，网络产生的那几十毫秒的“抖动”，足以产生“刻舟求剑”般的效果。

3. 其他技巧

要想使模型的收敛速度尽可能快，还有一些训练技巧可以尝试。

（1）使用灰度图

用灰度图代替原始输入的三通道图片是一个好办法，因为相同大小的灰度图比原始的三通道图片少 2/3 的输入数据。

可以考虑使用 OpenCV 的相关功能来实现，下面给出一个示例。

运行如下 OpenCV 的 Python 程序，看一下效果。

```
import cv2
from PIL import Image
import matplotlib.pyplot as plt

path = '/home/gao/SerpentAI/datasets/collect_frames_for_context/game/frame_15322272
92.7303226.png'

#以灰度图模式读取
img = cv2.imread(path, 0)

ret,thresh1 = cv2.threshold(img,127,255,cv2.THRESH_BINARY)
ret,thresh2 = cv2.threshold(img,127,255,cv2.THRESH_BINARY_INV)
ret,thresh3 = cv2.threshold(img,127,255,cv2.THRESH_TRUNC)
ret,thresh4 = cv2.threshold(img,127,255,cv2.THRESH_TOZERO)
ret,thresh5 = cv2.threshold(img,127,255,cv2.THRESH_TOZERO_INV)
titles = ['img','BINARY','BINARY_INV','TRUNC','TOZERO','TOZERO_INV']
images = [img,thresh1,thresh2,thresh3,thresh4,thresh5]
for i in range(6):
  plt.subplot(2,3,i+1)
  plt.imshow(images[i],'gray')
  plt.title(titles[i])
  plt.xticks([]),plt.yticks([])
plt.show()
```

如图 13-21 所示，对一幅三通道图片，可以进行二值化、反二值化等操作，每种操作都可以理解为降维。在这 6 幅图片里，对哪一幅，能用肉眼分辨出自己的飞机和敌人的飞机、自己打的子弹和敌人打的子弹？如果能用肉眼分辨，就说明图片中携带的信息没有发生丢失，就可以将这幅图片作为 State 输入模型。

需要注意的是，OpenCV 对于图片的处理和通道的保存方式，与 Python 默认采用的方式不一样。在 Python 里面，默认使用 R、G、B 三个通道，而 OpenCV 使用的是 B、G、R 三个通道，因此在存取多通道图片的时候要格外小心。用 OpenCV 保存一幅图片试试看——你会发现，“loser”居然从红色变成了蓝色。

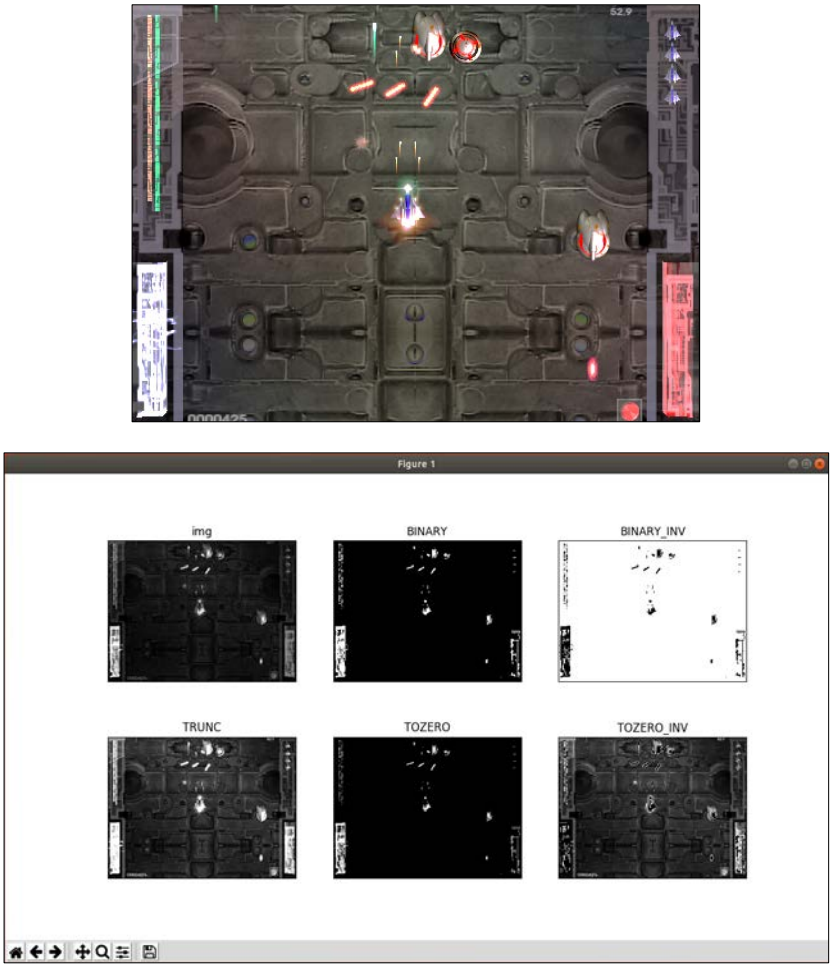


图 13-21　灰度图效果对比

（2）分辨率压缩

对于输入游戏的图像来说，如果只想对其进行有限的“降维”处理，SerpentAI 确实提供了原生的功能（如图 13-22 所示）。

在 GameFrame 类里面，有如下可以直接获取的属性内容。

- self.frame：全分辨率图像数据。
- self.half_resolution_frame：1/4 分辨率图像数据（宽度除以 2，高度除以 2）。
- self.quarter_resolution_frame：1/8 分辨率图像数据（宽度除以 4，高度除以 4）。
- self.eighth_resolution_frame：1/16 分辨率图像数据（宽度除以 8，高度除以 8）。
- self.grayscale_frame：灰度图像，全分辨率图像数据。

如果要使用 GameFrame 类，可以执行如下命令。

```
def handle_play(self, game_frame)
```

直接使用 `game_frame.half_resolution_frame` 这样的句柄，`self.frame` 就能拿到原始大小的帧数据，而 `game_frame.half_resolution_frame` 能拿到原始大小 1/4 的帧数据。

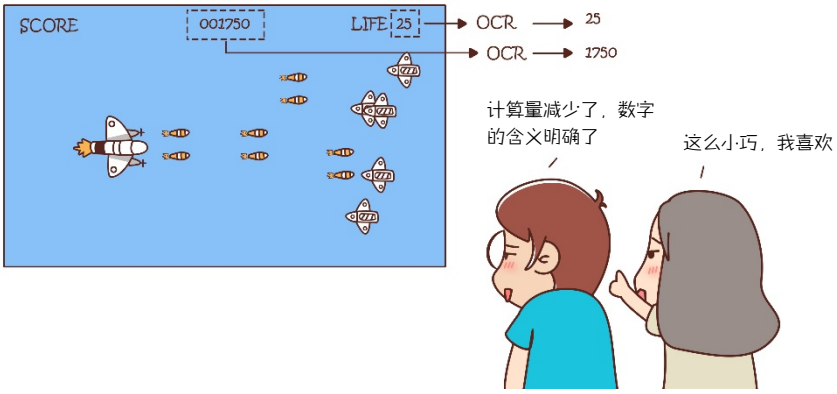


图 13-22　减少计算量

13.4　本章小结

我们在本章中接触了 SerpentAI 这个开源项目。应该说，SerpentAI 是一个比较“给力”的游戏代理框架，可以应用的场景也不限于游戏——任何 App（只要是有界面的）几乎都可以用来训练。这样一来，我们的“玩具”就多了起来。

需要强调的是，SerpentAI 有很多不足。在我的实践中，就发现它在 Windows 环境中对中文的支持不是很好，例如对中文窗口的识别有时无法通过（在 Ubuntu 环境中还没做过相关测试）。

SerpentAI 是一个非常优秀的项目，值得大家持续关注。SerpentAI 的官方文档，也应该成为我们的第一手阅读材料。在未来，相信 SerpentAI 会成为一个功能更为强大的、普适性更好的 App 代理工具——不仅在训练游戏上，在很多其他无人值守的应用中也能派上用场。此外，SerpentAI 在效率方面比“按键精灵”高，在扩展能力方面比“按键精灵”强。

对于本章的实验代码，读者可以访问本书 GitHub 页面的 DRLPytorch/SerpentAI 目录获取。至于是在本地修改对应路径，还是直接从网上重新下载并安装 SerpentAI 再手动编写代码，根据你的习惯来决定就好。其实，这两种方法都是可以使用的。

第 14 章　案例详解

到这里，我们对目前大部分与强化学习相关的主流机器人的训练方案及辅助工程和工具都做了比较详细的介绍，希望能够对大家的日常开发和研究提供一些帮助。

在本章中，将介绍一些非常有趣的案例和业界公认的挑战难度较高的公开项目。相信每位想要研究深度强化学习的读者，心里都有一个梦想，就是希望通过自己的努力，早日亲手编写出一个可以解决复杂问题的机器人智能体。本章的内容你一定不能错过，因为它会给你的研发工作带来提示和指引。

14.1　AlphaGo

首先，我们就介绍最为知名的——不仅为业内人士津津乐道，所有人在谈论时都忍不住咋舌的——AlphaGo！

14.1.1　AlphaGo 的前世今生

应该说，AlphaGo 具有一定的划时代意义（如图 14-1 所示）。为什么这么说呢？

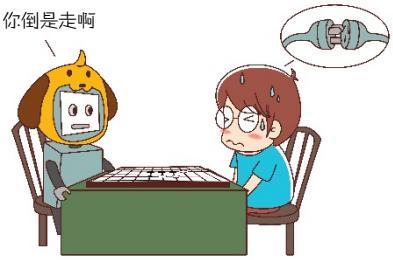


图 14-1　AlphaGo 比人更有耐心

首先，围棋 AI 几乎长期被全体人类视为人工智能挑战象牙塔上的明珠，从来都是各大人工智能研究机构所神往的研究方向，尤其在东亚各国的高校研究机构里更是如此。在理论基础和计算能力一直没有足够突破的时候，围棋 AI 的效果普遍不好。据一些懂行的朋友说，在 AlphaGo 成型之前，大部分围棋 AI 仅有业余一段的水平，最好的也不过业余二段左右[①]。所以，当谷歌宣布

① 中国的围棋段位分为专业段位和业余段位两种。专业段位从低到高依次是专业初段、专业二段……专业八段、专业九段。业余段位分为级和段两个阶段，水平从低到高依次是十级、九级……二级、一级、业余一段、业余二段……业余七段、业余八段。

自己的团队研究出了一个能够挑战人类的围棋机器人之后，所有人既兴奋、又怀疑。

其次，AlphaGo 作为深度学习和强化学习这样的尖端人工智能技术的结合体，第一次以如此平易近人的方式出现在世人面前，让我们第一次觉得 AI 技术触手可及。

2015 年 3 月 15 日，AlphaGo 以 4 比 1 的悬殊比分战胜韩国围棋大师李世石九段。当时就有人说，李世石赢的这一盘棋，应该是人类战胜计算机的最后一盘棋了。

自此，AlphaGo 成为围棋爱好者和人工智能爱好者茶余饭后的谈资。对这个陌生、可爱又有点可怕的机器人，还有人借助谐音，叫它“阿尔法狗”。当然，这只“狗”很厉害。

2015 年 10 月，AlphaGo 以 5 比 0 完胜欧洲围棋冠军、职业二段棋手樊麾。

2016 年 7 月 18 日，世界职业围棋排名网站 GoRatings 公布了当时最新的世界排名，谷歌旗下 DeepMind 研发的人工智能系统 AlphaGo 以 3612 分超越柯洁的 3608 分，成为新的世界第一。

2017 年 10 月，DeepMind 团队宣布，AlphaGo 的“小狗崽”AlphaGo Zero 问世了。AlphaGo Zero 可以“左右互搏”，仅用几天时间就打败了 AlphaGo。科技的进步真是令人叹为观止。

14.1.2 “深蓝”是谁

只要说到 AlphaGo，就不能不提另一个名字——深蓝（Deep Blue）。相信稍微有点年纪的读者都还有印象。

深蓝是 20 世纪 90 年代由 IT 业巨擘 IBM 设计研发的一套在当时有着超高计算能力的计算机，重 1270 千克，有 32 个微处理器，使用 AIX 操作系统[①]，每秒可以进行 2 亿次计算。

为了证明公司在研发层面的能力，IBM 决定让深蓝去挑战当时世界上最为强悍的国际象棋选手——加里·基莫维奇·卡斯帕罗夫[②]。

1996 年 2 月，深蓝和卡斯帕罗夫进行了第一回合的交手，但很可惜，它以 2 比 4 落败。比赛结束后，工程技术团队对深蓝进行了改进。1997 年 5 月，深蓝再次挑战卡斯帕罗夫——最终，深蓝以 2 胜 3 平 1 负的战绩战胜了当时世界排名第一的卡斯帕罗夫。可以说，从那一刻开始，在国际象棋领域，人类已经不可能战胜计算机了。

国际象棋和中国象棋的规则有些类似。同时，国际象棋在盘面的大小、棋子的数量、走棋的方式与限制等方面，与中国象棋都是同一个档次（或者说难度）的。

从棋子的种类和数量看：中国象棋分为黑、红两方，每方有“将”“士”“象”“马”“车”“炮”“兵”共 7 种棋子，棋子总数 16 个；国际象棋分为黑、红两方，每方有“王”“后”“象”“马”“车”“兵”共 6 种棋子，棋子总数也是 16 个（如图 14-2 所示）。

---

① AIX（Advanced Interactive eXecutive），在 IBM 商用计算机上使用的 UNIX 操作系统。

② 加里·基莫维奇·卡斯帕罗夫（Гарри Кимович Каспаров），苏联和俄罗斯国际象棋棋手，国际象棋特级大师。1963 年生于阿塞拜疆首都巴库。

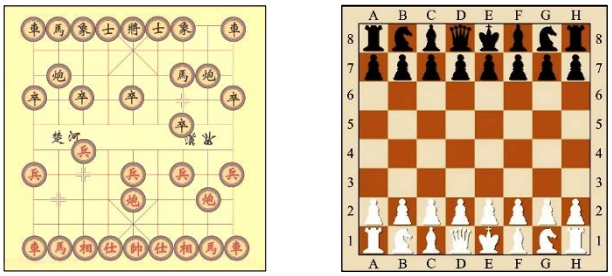


图 14-2　中国象棋和国际象棋的棋盘

从棋盘的大小看：中国象棋由横 9 纵 10 共 90 个交叉点组成盘面；国际象棋由横 8 纵 8 共 64 个方格组成盘面。

从棋子行走的方式和限制条件看，在中国象棋里：

- “将”走上、下、左、右各一格，且不能出宫；
- “士”走左上、左下、右上、右下，且不能出宫；
- “象”走“田”字，不能过河；
- “马”走“日”字，但前进时正前方不能有棋子，否则会算作“拌马腿”；
- “车”可以纵、横无限制走；
- “炮”可以纵、横无限制走；
- “兵”每次可以前进一格，过河后可以向前、左、右三个方向走，但不能后退。

在国际象棋中：

- “王”相当于中国象棋的“将”，可以上、下、左、右各走一格；
- “后”可以纵、横、斜走，无限制格数（这是国际象棋里面威力最大的棋子了）；
- “象”可以斜方向走，无限制格数；
- “马”走的位置，类似“日”字，只不过要先前进一格，再向斜方向前进一格（如图 14-3 所示）；
- “车”可以纵、横走，无限制格数；
- “兵”一次前进一格，只能击杀处在斜前方一格的敌人。

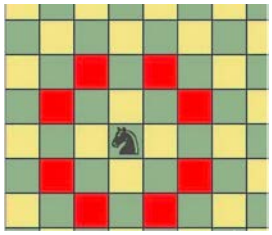


图 14-3　国际象棋中“马”的走法

尽管在国际象棋里面还有很多规矩，包括“王车易位”“兵升变”等，但业界仍普遍认为中国象棋和国际象棋在人工智能方面的研究难度是基本相当的。于是，人工智能领域似乎只剩下围棋可以挑战了。

14.1.3　围棋到底有多复杂

围棋是从中国古代流传下来的一种博弈游戏，在春秋战国时期就有记载。

围棋的规则非常简单。然而，简单的游戏玩起来往往非常复杂。围棋的复杂不在于规则本身，而在于变化的丰富。

如图 14-4 所示，围棋棋盘是 19 路 × 19 路的，共有 361 个交叉点，在每个交叉点上都可以放置一枚棋子。黑棋有 181 枚棋子，白棋有 180 枚棋子，按照先黑后白的方式交替落子，当最后一手（黑棋）落子的时候，正好填满这 361 个交叉点。

图 14-4　围棋盘面

在国际比赛中，围棋的规则是非常复杂的，包括赛场纪律、计时、读秒等。而对于我们这些初学者来说，只要了解通过盘面判断胜负的标准，并对其有一个感性的认识，就可以了。

1. 获胜条件

因为围棋的博弈内容比较复杂，所以，内行棋手为了沟通方便，会使用术语来表示一些特殊的棋局和招式。你经常会听到的有“气”“点眼”“小飞守角”“星小目”“征子”“渡”“劫材”“七死八活”“双”“尖冲”“刺”“立二拆三”“扳”“脱先”“高夹”等深奥的名词。不过，对于胜负的判断，还是相对比较简单的。

其实，计算机对这些术语是没有辨别能力的，或者说，计算机是不需要辨别这些信息的。因为对计算机来说，这些信息只是从一个状态到另一个状态的 Transition。

除了中途有人直接认输的情况以外，围棋棋局通常要在双方都同意结束时才可以结束，此时终局即告成立。在这个时候，要做的事情就是“算账”了。双方要根据规则来“打扫战场”，即对盘面上的棋子进行判断——哪些是活棋，哪些是死棋。如果是活棋，就保留下来；如果是死棋，就把它从盘面上拿掉。

双方活棋之间的空点各得一半。

棋盘总点数的一半（180.5 点）为归本数。一方总得点数超过此数为胜，等于此数为和，小于此数为负。换句话说，下围棋的最终目的是看谁的地“圈”得多，而不是像象棋那样，击杀主将或者击杀对方棋子多的一方获胜。

关于如何计算，也有不同的规矩可循。

围棋界通用的终局判断胜负的方法有如下三种。

- 数子法：在我国主要应用这种方法。对所围范围内的子和空，都要数子，每个交叉点计 1 子。中国数子法目前贴 $3\frac{3}{4}$ 子。终局时，双方将棋盘上所有交界处的空点占完，即表示收完单官。当黑棋总数超过 $\frac{361}{2}+3\frac{3}{4}=184\frac{1}{4}$ 子时，才算黑方胜，否则为白方胜。黑棋为什么要贴子呢？这是因为，在围棋领域，执黑先行的一方会占有一些优势，这一点在棋力相当的对弈者之间会体现得比较明显。为了将这种优势尽可能拉平，让双方不要因为执黑先行而产生棋力对比上的误差，特意让执黑棋的一方贴补执白棋的一方 $3\frac{3}{4}$ 子，以示公平。在其他回合制棋类游戏中也有类似的规则。例如，象棋的“红先黑后”也会导致一方占优，只不过由于象棋棋子数量太少，别说让个“车”，让个“卒”都能让局势大不一样。此外，在有双活时，有眼双活的，眼和活棋计为自己的子，公气点双方各计半子。在让子时，通常黑棋不贴目。
- 数目法：数目法主要在日本和韩国应用。先将提吃对方的棋子填回对方的空内，然后点目（自己势力范围内的空，每个点计 1 目）。数目法贴 6.5 目，与数子法类似，只是计数单位不同。日本采用的规则是不收单官，韩国采用的规则需要收完单官。对双活的棋，如果使用日本数目法，无论何种双活，双方均不计目。对让子棋，一般黑棋不贴目。
- 应氏计点制规则[①]：也称应氏数子法、应氏规则，在“应氏杯”及相关比赛中使用。以数子法规则为基础，双方收完单官后，将棋罐内的棋子全部填到棋盘的空点处，直到填不下为止，用剩余的棋子数减去贴点数，即得到胜负结果。应氏规则贴 8 点。除此之外，应氏规则中有关于计时罚点的说法，即采用超时罚点制，以罚点代替读秒。“应氏杯”赛事规则一般为“每方 3.5 小时+3 次 35 分罚点”，即用完 3.5 小时后，超时 35 分钟内罚 2 点，超时 35 ~ 70 分钟罚 4 点，超时 70 ~ 105 分钟罚 6 点，超时 105 分钟判负。应氏规则的操作相对比较复杂，这主要是应氏规则希望考虑更多的因素，从而使评判更全面、更公平所致。

2. 复杂度估算

围棋的棋盘为 19 纵 19 横的直线相交盘面，盘面上一共有 361 个交叉点。相传，围棋是上古

① 与数子法、数目法并列的三大围棋规则之一，由应昌期先生创立。采用这个规则的比赛有“应氏杯”世界职业围棋锦标赛、世界青少年围棋锦标赛、世界大学生围棋锦标赛及“倡棋杯”中国职业围棋锦标赛等。应昌期（1917 年 10 月—1997 年 8 月），祖籍宁波市江北区慈城镇，台湾地区金融界、实业界著名人士，宁波市荣誉市民，应氏围棋计点制的创造人，被誉为“黑白世界的诺贝尔”。

五帝之一的尧[①]为了教育自己的儿子丹朱所创造的。围棋在中国有悠久的历史，在隋唐时期经由朝鲜传入日本，在东亚诸国乃至全世界都有相当多的爱好者。

多年以来没有人工智能领域的机器人模型能够征服围棋，主要还是围棋的复杂度与计算能力之间的差距所致。

有关复杂度的计算，我做一个简单的说明，好让大家心里有数。

首先是关于合法局面数量的估计，如图 14-5 所示。由于棋盘上的每个位置都可以出现“空”“白”“黑”三种情况，对于边长为 n 的棋盘，共有 3^{n^2} 种局面（相当于整个空间数量的上限）。但是，其中只有一部分局面是合法的，而相当一部分局面是不可能出现的（不是从逻辑上不能出现，而是规则不允许出现）。有多少种局面呢？一位名叫 John Tromp 的荷兰计算机科学家经过计算，推导出了一个递归公式 $L(m,n)$，用来计算一个长为 m、宽为 n 的矩形棋盘的合法局面数量。在 2016 年，John Tromp 计算出了 $L(19,19)$ 的大小，也就是现在标准围棋棋盘的大小。

盘面大小 $n\times n$	3^{n^2}	合法的比例	L（合法局面）（A094777）[11]
1×1	3	33.33%	1
2×2	81	70.37%	57
3×3	19,683	64.40%	12,675
4×4	43,046,721	56.49%	24,318,165
5×5	847,288,609,443	48.90%	414,295,148,741
9×9	$4.43426488243\times10^{38}$	23.44%	$1.03919148791\times10^{38}$
13×13	$4.30023359390\times10^{80}$	8.66%	$3.72497923077\times10^{79}$
19×19	$1.74089650659\times10^{172}$	1.20%	$2.08168199382\times10^{170}$

图 14-5　围棋盘面空间复杂度计算[②]

根据数据基本可以断定，随着棋盘尺寸的增加，合法的局面数量虽然会增加，但所占比例会降低。对于 1×1 的棋盘，为什么合法局面只有 1 个呢？在黑先白后的规则下，对 1×1 的棋盘，只要黑方先行，必然只有一个局面——只有 1 粒黑子。而对 19×19 这么大的棋盘，合法局面的数量达 2.08×10^{170} 个——比目前可宇宙内可观测的所有原子数量的估值 10^{80} 高 90 个数量级！什么叫天文数字，这就叫天文数字。这相当于，围棋棋盘上最多有 2.08×10^{170} 个合法的状态 S 的取值——想想都“肝儿颤”。

整棵博弈树的复杂性作为另一个数学问题，也有人进行了研究。这就是我们所说的整棵树的复杂度，即本书前面反复提到的用 DP、MC 进行评估时看到的状态转移树。

① 尧（约前 2377—前 2259 年），姓伊祁，号放勋，古唐国人（今山西临汾尧都区）。中国上古时期方国联盟首领、“五帝”之一。

② 引自维基百科：https://en.wikipedia.org/wiki/Go_and_mathematics。

一位名叫 Victor Allis[①]的计算机科学家提出了一个观点：职业棋手之间的对弈一般在 150 手[②]左右，平均每手大概有 250 种不同的盘面选择，也就是说，这棵树应该有 150 层，每一层的每一个节点下面都有 250 个节点——一棵极高、极茂盛的树（如图 14-6 所示）。按照这样的估算原则，这棵树最后一层的节点数应该是多少呢？因为在这棵树上，从任何一个最后一层节点退回树顶端的这条路，都代表一个棋局，所以，如果把这个值估算出来，就意味着能够大体知道有多少种不同的棋局了。结果应该是 250^{150}，大约在 10^{360} 这个数量级。

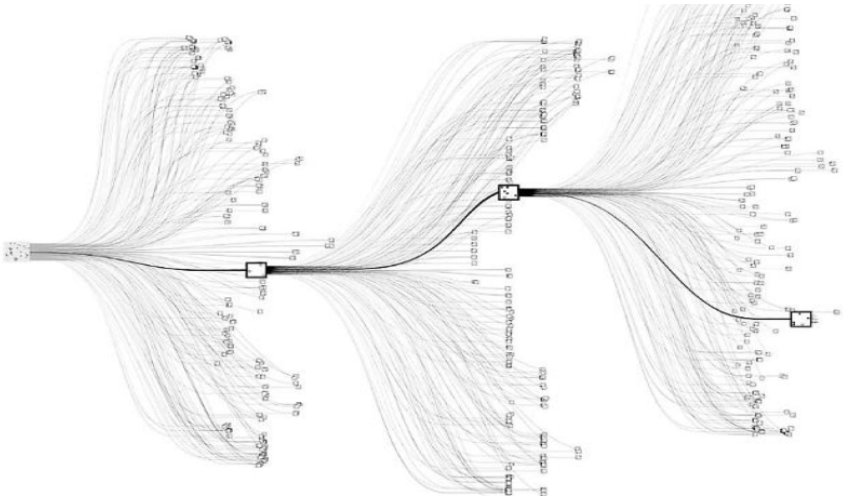


图 14-6　围棋盘面树搜索展开示意图

推广一下这种计算方法。如图 14-7 所示，如果每手棋都有 N 种选择，一局中共有 L 手棋，那么棋局总数就是 N^L。

棋盘大小	交叉点数N	$N!$	平均对局手数L	N^L	最长对局手数	棋局数量下限	棋局数量上限
2×2	4	24	3	64		386,356,909,593[14]	386,356,909,593
3×3	9	3.6×10^{5}	5	5.9×10^{4}			
4×4	16	2.1×10^{13}	9	6.9×10^{10}			
5×5	25	1.6×10^{25}	15	9.3×10^{20}			
9×9	81	5.8×10^{120}	45	7.6×10^{85}			
13×13	169	4.3×10^{304}	90	3.2×10^{200}			
19×19	361	1.0×10^{768}	200	3×10^{511}	10^{48}	$10^{10^{48}}$	$10^{10^{171}}$
21×21	441	2.5×10^{976}	250	1.3×10^{661}			

图 14-7　围棋博弈树的复杂性[③]

① Victor Allis（1965 年 5 月—），荷兰计算机科学家。
② 一位棋手下一步棋叫作一手，一回合则是二手。
③ 引自维基百科：https://en.wikipedia.org/wiki/Go_and_mathematics。

我们只看 19×19 这一行数据（这个图表对比了以不同的方法估算出来的树的复杂性）。例如，用 $N!$ 和 N^L。N^L 刚才我们解释过了，$N!$ 则认为这 *N* 种不同的选择其实是一个有限集合，如果第一步有 *N* 种选择，第二步就只有 $N-1$ 种了，第三步则是 $N-2$ 种……按照这种逻辑，用排列组合的方式计算，就是 $N!$。和 N^L 的方式一样，这也是一种估算值。

无论哪种估算值，都是一个巨大的数字。一个数量级就表示 10 倍计算能力的差距。这时，单纯堆积硬件是不足以解决问题的，需要采用很多的技巧。不过，这么说不代表硬件资源不重要，恰恰相反——重要！非常重要！把这一章的案例读完你就会知道，每一个项目的成功，都离不开计算机处理能力的支持。

14.1.4 论文要义

在本节中，我们介绍一篇著名的有关 AlphaGo 原理的论文——*Mastering the Game of Go with Deep Neural Networks and Tree Search*，它发表在 2016 年的 2 月的 *Nature* 上。这篇论文的领衔人物就是大名鼎鼎的 David Silver[①]。这篇论文是公开的，可以下载。把它打印出来，足足有 38 页 A4 纸。下面，我帮大家把这篇论文读“薄”一些。

在这篇论文中，穿插介绍了 AlphaGo 的整个模型结构和训练过程的完整思路。

1. SL Policy Network

对一个棋类博弈问题来说，首先要决定建模的方向。用什么样的方式来实现？我们可能会马上想到使用神经网络（先进一点，就用 DQN）。使用神经网络的思路是不错的，因为神经网络的空间非常大。如果使用传统的 Q-Learning 查表方式，维护整个状态集合的成本就会很高，不论是保存还是做搜索，都是不可行的——Q-Learning 不行，传统的 MC 和 DP 恐怕也不行。从这个角度看，神经网络应该是首选，即用正向传播来代替传统的方法。

此外，如何实现冷启动是一个问题。如果让一个机器人在任何“基础知识”都没有的情况下进行试探，那就是去试探整棵合法的空间树。对这棵树的规模，我们已经进行了估算——一个天文数字。这就意味着，在相当长的时间内，这棵树上某个状态的估值 $V(s)$ 是很难得到的，而这对收敛速度有很大的影响。为了让 AlphaGo“学”得快一些，在整个训练过程的第一个阶段，采用了一种叫作模仿学习（Imitation Learning）的方式。所谓模仿学习，看上去更像监督学习，在论文里称作“Supervised Learning of Policy Network”（SL Policy Network），也就是策略网络的监督学习（如图 14-8 所示）。

先用监督学习的方式训练一个叫作 p_σ 的网络。这个网络的训练比较简单，训练样本就是围棋大师们的行棋样本。论文中说，这些数据都来自 KGS[②]，我们就理解为大师们对弈的棋谱吧。

---

① David Silver（1976 年—），加拿大阿尔伯塔大学（University of Alberta）博士，著名强化学习专家，就职于 DeepMind 团队，领导研发 AlphaGo。

② 一个免费的围棋服务器，http://www.gokgs.com/。

也就是说，无论是黑方还是白方，都是高质量的行棋内容。将这些内容分解之后，就得到了一个盘面的信息，以及一个对应于此盘面的高质量的行棋点位。这样的信息简直是太好了，一下子就让我们想到了 MNIST。一个盘面信息可以理解为一幅手写图片，一次行棋的内容就是一个在 19 × 19 的点位上的位置信息——典型的分类预测。利用这样的信息，就可以直接进行离线训练了。在这里训练了一个有 361 个分类的图片分类器。这个训练出来的网络 p_σ 就是 SL Policy Network，用来预测在一个盘面下一位人类围棋大师将在何处落子。这是一个典型的监督学习过程，如果 MNIST 能明白你是怎么做分类的，那么这个过程将一点新鲜感都没有。

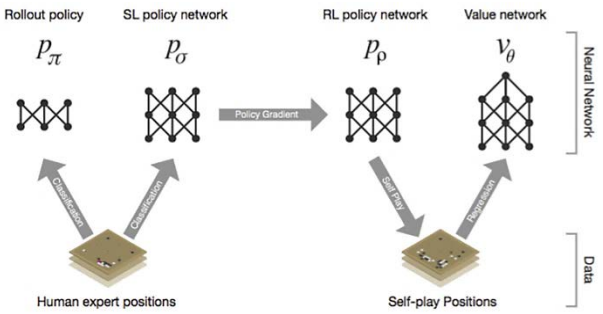


图 14-8　策略网络示意图

整个 SL Policy Network 就是一个典型的 CNN，如图 14-9 所示。

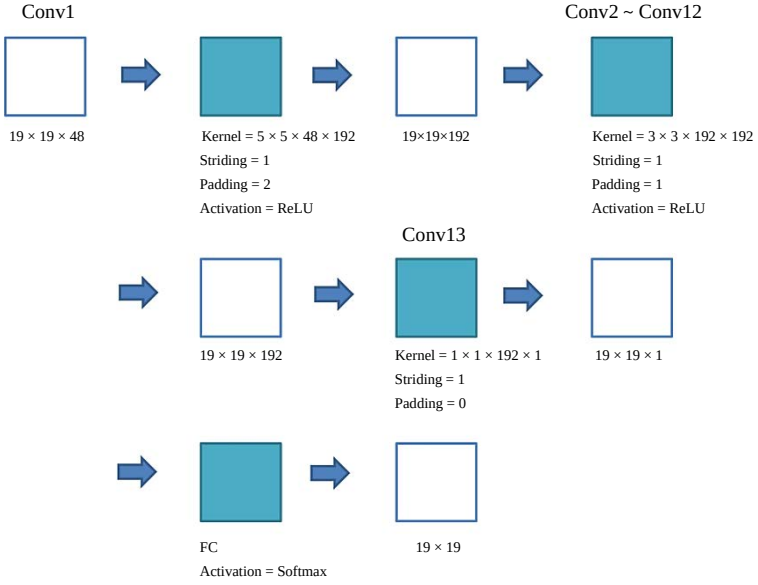


图 14-9　SL Policy Network 结构示意图

这是一个深度卷积神经网络。输入是一个描述盘面信息的 $19 \times 19 \times 48$ 的张量。后面是 13 层卷积，Conv1 是一个规格，从 Conv2 到 Conv12 是一个规格，Conv13 是一个规格——大家自己看参数就可以了。

卷积神经网络这东西，开句玩笑，多一层少一层又能怎么样？但是，盘面为什么是 $19 \times 19 \times 48$ 的？还是要解释一下。

19×19 容易理解，就是 361 个点。每个点上的 48 指的是 48 个特征描述。都有什么呢？每个特征就相当于一个通道的描述，棋子使用了 3 个通道，己方子、对方子、空白各 1 个通道（如图 14-10 所示）。这 3 个通道是客观存在的，其他通道是人为添加的。

特征	平面数量	说明
棋子颜色	3	自己的棋子、对手棋子、空白位置
1	1	全1平面
轮次	8	每个落子过后经过的轮次
气	8	每个落子气的数量（邻近空的点）
打吃	8	对手被打吃的数目
被打吃	8	自己被打吃的数目
落子后的气	8	每个落子刚落之后气的数量
征子有利	1	落子是否征子有利
征子逃脱	1	落子是否征子逃脱
合法性	1	落子是否合法并且没有填自己的眼
0	1	全0平面
颜色	1	是否当前是执黑

Feature	# of planes	Description
Stone colour	3	Player stone / opponent stone / empty
Ones	1	A constant plane filled with 1
Turns since	8	How many turns since a move was played
Liberties	8	Number of liberties (empty adjacent points)
Capture size	8	How many opponent stones would be captured
Self-atari size	8	How many of own stones would be captured
Liberties after move	8	Number of liberties after this move is played
Ladder capture	1	Whether a move at this point is a successful ladder capture
Ladder escape	1	Whether a move at this point is a successful ladder escape
Sensibleness	1	Whether a move is legal and does not fill its own eyes
Zeros	1	A constant plane filled with 0
Player color	1	Whether current player is black

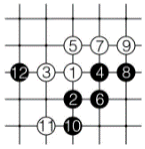
Extended Data Table 2: **Input features for neural networks.** Feature planes used by the policy network (all but last feature) and value network (all features).

图 14-10　输入特征的中英文对照

我们以其中一个为例，“轮次”就是手数，即第几手棋的概念。这相当于把轮次（论文中称作“Turns Since”）向量做了一个二值化的表示，即这一步距离现在有多久。如图 14-11 所示，P1 通道表示“现在”，因为第 12 手的位置是距离现在最近的，其时间距离就是“1”，所以，第 12 手这个点就表示在 P1 上第 11 手与现在在时间上的距离是“2”（这个对应的点就出现在 P2 通道上），P8 通道则表示与现在的时间距离“8”及以前的落子，所以，第 1 手到第 5 手都在上面了——就是这样。

照理说，使用“黑子”“白子”“无子”三个通道已经足够描述整个盘面的信息了。然而，这些多出来的通道中包含了人们对状态的特征进行的一些预处理内容，这也是在 DRL 中经常使用的技巧。使用这些人为设定的特征，看上去是一种升维手段，实则是一种降维手段。这些新的维

度，在人类看来，有不少是用来“帮助”计算机理解状态中的高级特征或者抽象特征的。虽然深度学习在特征提取方面有先天的优势，不过，让计算机在浩如烟海的“像素”级别的信息中通过卷积核不断进行甄别的过程需要海量的数据——即便在 GPU 集群环境中，也要经过“旷日持久”的训练才能完成。加入这些高层特征，以及重要的、有代表性的维度特征，相当于为计算机提供了条件概率判断的熵减条件。因此，这样一种技巧，不仅在 AlphaGo 的训练中经常使用，在其他游戏机器人的训练中也经常使用。




```
[0 0 0 0 0]
[0 0 0 0 0]
[1 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
```
P1
```
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 1]
[0 0 0 0 0]
[0 0 0 0 0]
```
P5

```
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
[0 1 0 0 0]
```
P2
```
[0 0 0 0 0]
[0 0 0 1 0]
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
```
P6

```
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 1 0 0]
```
P3
```
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 1 0]
[0 0 0 0 0]
```
P7

```
[0 0 0 0 0]
[0 0 0 0 1]
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
```
P4
```
[0 0 0 0 0]
[0 0 1 0 0]
[0 1 1 1 0]
[0 0 1 0 0]
[0 0 0 0 0]
```
P8

图 14-11　轮次盘面示例

如图 14-12 所示，是在不同参数设置和输入的情况下训练出来的网络的表现。“Filters”表示卷积核的数量，“Features”表示特征通道输入的数量。对这些在不同超参数下的模型的表现进行对比，也是不得已的，因为超参数是没办法“学”出来的——只能比来比去，“比”一个好一点的出来。在这个有 361 个分类的网络中，最高的正确率仅为 57%——其实还算可以，毕竟有 360 个分类错误的机会呢……

Architecture			Evaluation				
Filters	Symmetries	Features	Test accuracy %	Train accuracy %	Raw net wins %	*AlphaGo* wins %	Forward time (ms)
128	1	48	54.6	57.0	36	53	2.8
192	1	48	55.4	58.0	50	50	4.8
256	1	48	55.9	59.1	67	55	7.1
256	2	48	56.5	59.8	67	38	13.9
256	4	48	56.9	60.2	69	14	27.6
256	8	48	57.0	60.4	69	5	55.3
192	1	4	47.6	51.4	25	15	4.8
192	1	12	54.7	57.1	30	34	4.8
192	1	20	54.7	57.2	38	40	4.8
192	8	4	49.2	53.2	24	2	36.8
192	8	12	55.7	58.3	32	3	36.8
192	8	20	55.8	58.4	42	3	36.8

图 14-12　参数与训练结果对照

SL Policy Network 使用了 2940 万个盘面（约 16 万个完整对局，且玩家都是六段到九段的专业棋手），其中 100 万个盘面用于测试，2840 万个盘面用于训练。在训练时使用了 50 个 GPU，耗时 3 周左右。

在这种情况下，论文的作者认为，SL Policy Network 这个模型已经能够比较成功地识别一些定式或者形态了。换句话说，已经可以靠这种监督学习方法，得到第一代能够模仿这些六段到九段专业人士的行棋常用套路的模型了。

2. Rollout Policy Network

有了 SL Policy Network，其实就可以考虑用它来设计产生样本的工具了。然而，这样的网络实在太“深”了——有超过 380 万个参数。在 128 个通道的卷积核中进行一次正向传播，至少需要 2.8 毫秒；如果有 192 个通道，则需要 4.8 毫秒。时间实在是太长了，每秒只能产生 200 ~ 300 个样本，因此，需要重新训练一个速度更快的模型来做相同的事情。

在相同的硬件环境中，如果想提高网络的正向传播速度——要么使用一些对网络有损的压缩技术（这类问题有一批数据科学家在专门研究，这类课题的英文名称是“Model Compression”或“Model Acceleration”，基本是同领域的研究范畴，压缩意味着计算量减小且可以加速）；要么只能采用一些结构更为简单的模型以降低计算的复杂度。在一定程度上，前者和后者的目的是非常相近的。这样，人们就有了设计一个简化的模型的动机，而代价是降低一定的精度。这个简化的模型就是 Rollout Policy Network。

真是遗憾，我用金山词霸网络版也没有查出“rollout”在这个句子中该怎么解释，反正不能翻译成“首次展示策略网络”这么奇怪的词组，如图 14-13 所示。从作用上看，就是为了快速落子、快速产生样本，为计算估值做准备，如图 14-14 所示。

图 14-13 用金山词霸查询“rollout”的意思

Feature	# of patterns	Description
Response	1	Whether move matches one or more response features
Save atari	1	Move saves stone(s) from capture
Neighbour	8	Move is 8-connected to previous move
Nakade	8192	Move matches a *nakade* pattern at captured stone
Response pattern	32207	Move matches 12-point diamond pattern near previous move
Non-response pattern	69338	Move matches 3×3 pattern around move
Self-atari	1	Move allows stones to be captured
Last move distance	34	Manhattan distance to previous two moves
Non-response pattern	32207	Move matches 12-point diamond pattern centred around move

图 14-14　Rollout Policy Network 的 Feature 说明

Rollout 网络的输入特征是人为做的一些较高层次（或者说抽象一些）的特征提取。例如，“Response”指的是“应对”，也就是应对上一步走法；“Save Atari”不是“保存 Atari 游戏”的意思，它和雅塔力游戏一点关系都没有，这是日语“あたり”的英语写法，包括下面的“Nakade”，日语为“なかで”，它们分别指“叫吃”和“眼形”。不仅论文中的围棋术语都是用日语假名的罗马字表示的，在其他领域引用围棋术语时，也普遍使用日语的术语读音（例如，“AlphaGo”里的“Go”，就使用了日语中围棋“碁（ご）”的读音）。

这个 Rollout 网络被定义为 P_π，是用一个小型的线性 Softmax 模型实现的，精度大概只有 24.2%，也就是说，在超过 75% 的位置的预测不准确。这些不准确的情况，我觉得应该是预测出来的落子位置不太像大脑处于清醒状态的围棋大师（人都是会疲劳的，都有大脑不清醒的时候，围棋大师也不例外）下棋的路子所致，所以，不能说这样的落子是错的。而且，在 AlphaGo 成功出世的时候，我们才发现，这些看起来错误的落子，很多都是原来没有被发掘出来的具有极高价值的走法。这个网络很厉害，走一步棋只需要大约 2 微秒，也就是说，每秒可以生成 50 万个样本。这可是 3 个数量级的提升啊！

3. RL Policy Network

当前两步的网络训练铺垫完成后，就可以开始正式进行深度强化学习的训练了，这部分网络叫作“RL Policy Network”（Reinforcement Learning Policy Network）。

如果要训练 RL Policy Network，从理论上讲，可以采取的方法有 DQN（这是时间差分的方法）、DDPG（这是梯度策略的方法）、MCTS（这是纯粹值估计的方法）。不过，不管采用哪种方法，目的都是确定 RL Policy Network 这个机器人在环境中获得的奖励值 Reward（因为任何一种强化学习方法都要把状态函数的值计算得尽量准确）。在 AlphaGo 这样一个具体的场景中，用哪个方法好呢？

想想看：是策略梯度更合适，还是离散的估值更合适呢？如果要使进化“柔和”一些，策略梯度法的效果比纯粹的时间差分法要好一些。而且，现在我们就是要训练一个网络来做这件事，不用考虑查表的问题。因此，让策略梯度网络输出一个动作的概率分布应该是一个不错的选择，如图 14-15 所示。

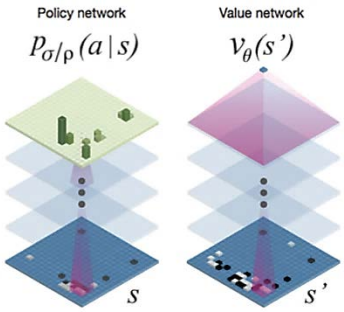


图 14-15 Policy Network 和 Value Network

定义一个网络 $p_\rho(a|s)$，让它的结构和 $p_\sigma(a|s)$ 相同。这个网络就是最终输出策略的网络。同时，我们需要一个估值网络 $v_\theta(s')$ 来进行估值计算。这就是策略梯度的套路，看上去就是 Actor-Critic 的模式。这两个网络分别应该怎么去训练呢？如图 14-16 所示。

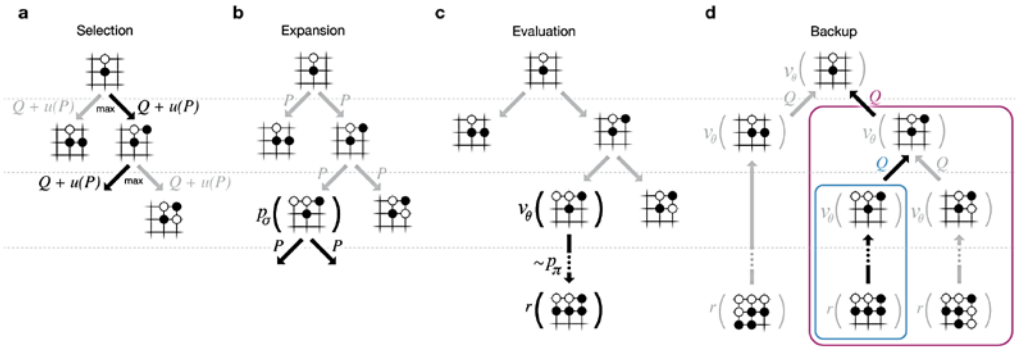


图 14-16 用 MCTS 来做估值

有关估值的方法，前面我们讲了不少，例如 DP、MC、TD，都是用来做估值的。AlphaGo 用 MCTS 来做估值，即通过深度优先的遍历方式来做估值，然后通过不断迭代来计算估值的数学期望值。对于一个状态的估值，就训练一个 $v_\theta(s')$ 网络来做估值，而这就是一个纯粹的用卷积神经网络实现的回归问题了。输入为棋盘的特征，输出为胜率——对，就是一个概率值，一个 0 到 1 的分布——当然越高越好。这部分数据集是通过 SL Policy Network 和 RL Policy Network 共同生成的（如图 14-17 所示）。

大概过程是：生成 3000 万个盘面，随机计算某个时间点 U；从第 1 步到第 $U-1$ 步，都使用 SL Policy Network 来生成；第 U 步随机走，第 U 步到第 T 步使用 RL Policy Network；分别计算输和赢的概率；将状态 s_{U+1} 和胜负 z_{U+1} 放入数据集 (s_{U+1}, z_{U+1})，表示这个盘面最终胜利与否，即预测

$$v^p(s) = E[z_t|s_t = s, a_{t\ldots T}{\sim}p].$$

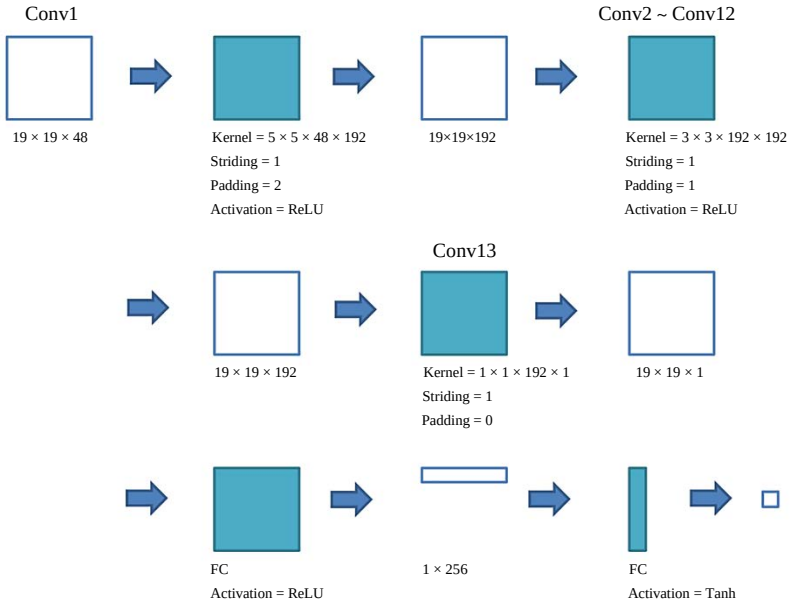


图 14-17　Value Network 结构示意图

论文中也说了，这部分训练是为了把 MCTS 和深度学习结合在一起。AlphaGo 使用了异步多线程搜索。针对策略网络和估值网络的更新运算，AlphaGo 在 GPU 上做了并行的梯度下降。最后一个版本的 AlphaGo 使用了 40 个搜索线程并行做 MSTC，其硬件环境包括 40 个 CPU 和 8 个 GPU。这看起来可能不足为奇，因为就算使用当时的顶级配置，花费也仅为几十万元人民币——对 AlphaGo 这样的项目来说，已经很经济了。不过，AlphaGo 团队为了加速训练，还做了一个分布式版本，使用了 40 个搜索线程，硬件环境包括 1202 个 CPU 和 176 个 GPU……

网络的大部分结构和 SL Policy Network 差不多，区别是，这里最后的输出是一个 Tanh 函数——看来得分还能是负数（从 –1 到 1）。

策略网络 $p_\rho(a|s)$ 的训练和普通 Actor 的训练方式相似，就是一个对状态策略进行梯度提升的过程，优化的参数是 θ，优化的函数应该是给定状态在某个策略下的价值的数学期望值

$$J(\theta) = V^{\pi_\theta}(s) = E_{\pi_\theta}[s].$$

$E_{\pi_\theta}[s]$ 的值应该是通过 MCTS 算法的不断搜索而估算出来的。根据论文中的说法，对每次搜索得到的 Reward，如果未分胜负就是 0，如果胜利就是 1，如果失败就是 –1。梯度计算也不难，根据梯度策略的更新公式就可以完成：

$$\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta \log_{\pi_\theta}(s,a) Q^{\pi_\theta}(s,a)].$$

在论文中有一个细节：

$$\Delta_\rho = \frac{\alpha}{n}\sum_{i=1}^{n}\sum_{t=1}^{T_i}\frac{\partial \log_{p_\rho}(a_t^i|s_t^i)}{\partial \rho}\left(z_t^i - v(s_t^i)\right).$$

这里说的是，对于 RL Policy Network 的梯度更新，每次更新的量是 Δ_ρ。但是，对于 Δ_ρ 的计算，在最后一项 $z_t^i - v(s_t^i)$ 中，$v(s_t^i)$ 是由 Value Network 提供的状态估值，z_t^i 是通过 MCTS 算法搜索得到的估值。每次进行这种求差的操作，都可以让 MCTS 的方差减小，从而帮助网络稳定地收敛。T_i 表示一个 Episode 的步数，n 表示 Episode 的个数。总的来说，这仍然是一种多次求策略取平均值的思路。

对 SL Policy Network、Rollout Network、RL Policy Network、Value Network 的含义，我们已经进行了详细的讨论，最后介绍一下整个训练过程。

①初始化 RL Policy Network，把 SL Policy Network 的参数赋值给 RL Policy Network，让它获得基本的行棋能力。

②RL Policy Network 会创造一个新的版本 N，使其与之前最新的版本 $N-1$ 对弈，并在这个过程中不断得到胜负判定信息和相应的 Transition 信息。

③在这个过程中，不断对 RL Policy Network 和 Value Network 进行迭代的梯度更新。

④每经过 500 次迭代，就把最近训练的 RL Policy Network 准备好，回到第②步，作为下一个版本的“假想敌”。如此反复。

RL Policy Network 每天要在 50 个 GPU 上训练 10000 个 Mini Batch，这样的速度是人类大脑无法企及的。

14.1.5　成绩

关于 AlphaGo 棋力的评价，如图 14-18 所示。

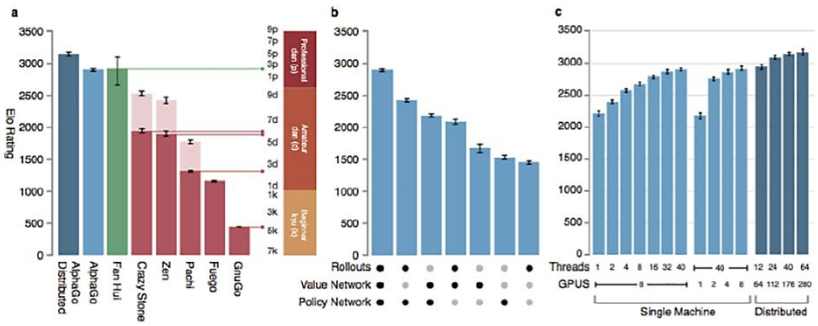


图 14-18　AlphaGo 的棋力

图 a 说的是：用 Elo[①]评分体系对不同的围棋软件和棋手进行评分，分布式 AlphaGo 的得分是 3100 分左右，相当于职业五段棋手；AlphaGo 和它战胜过的棋手樊麾，得分在 2900 分左右，相当于职业二段棋手（比起其他软件，还是高出了一大截）。

图 b 说的是：单机版 AlphaGo 在不同组件组合下的评分水平能达到什么程度。如果三个点都为黑色，就表示 Rollouts Network、Value Network 和 Policy Network 三个网络一起工作，这时能得到大约 2900 分。最差的是只使用 Rollouts Network 的情况，得分不足 1500 分。

图 c 说的是：搜索线程和 GPU 的数量增加对分数的影响。这个影响不是线性的。1 个 GPU 和 40 个搜索线程，得分为 2200 分左右；280 个 GPU 和 64 个搜索线程，大概有 3100 分的得分。

不管得分如何，后面的事情大家都知道了——2016 年 3 月 15 日，AlphaGo 以 4 比 1 的悬殊比分轻松击败韩国九段棋手、世界围棋冠军李世石。是的，AlphaGo 太可怕了，只要通电，它就能“上自习”。

2016 年 12 月 29 日晚到 2017 年 1 月 4 日晚，AlphaGo 以“Master”为注册名，在弈城围棋网和野狐围棋网与数十位人类顶尖围棋高手对战，取得了 60 胜 0 负的辉煌战绩。

2017 年 5 月 23 日到 27 日，在中国乌镇围棋峰会上，AlphaGo 以 3 比 0 的总比分战胜了当时排名世界第一的世界围棋冠军柯洁。在这次峰会期间，2017 年 5 月 26 日，AlphaGo 还战胜了由陈耀烨、唐韦星、周睿羊、时越、芈昱廷五位世界冠军组成的围棋团队。

至此，可以说，AlphaGo 在下围棋这件事情上“天下无敌”了。

14.1.6　开源项目

有不少爱好者自己对 AlphaGo 进行了实现并将其放到网上供大家学习。我在 GitHub 上找到了一个名叫“RocAlphaGo”的星比较多的项目（https://github.com/Rochester-NRT/RocAlphaGo）。从 8800 多颗星的情况来看，这个项目还是很靠谱的。

其页面第一行的内容是：“An independent, student-led replication of DeepMind’s 2016 Nature publication, ‘Mastering the game of Go with deep neural networks and tree search’.”意思很清楚：“这是一个独立的、由学生主导的、以 DeepMind 在 2016 年发布的 *Mastering the Game of Go with Deep Neural Networks and Tree Search* 为蓝本的副本。”也就是说，这是一段“野生”的 AlphaGo 代码，是由其作者根据相关论文实现的。

AlphaGo 取得的骄人成果，与其说是一个目标，不如说是一个起点。此后，DeepMind 团队又进行了一系列的研究，制造了一个终极“武器”——AlphaGo Zero——比 AlphaGo 还厉害。

---

① Elo Rating，一种用来评价零和游戏玩家等级与能力的评分体系。可以用来评价体育竞技排名，常用在棋牌游戏等竞技场合。中国九段棋手、世界冠军柯洁在巅峰时得到了 3600 多分，韩国九段棋手李世石在巅峰时得到了 3500 多分。

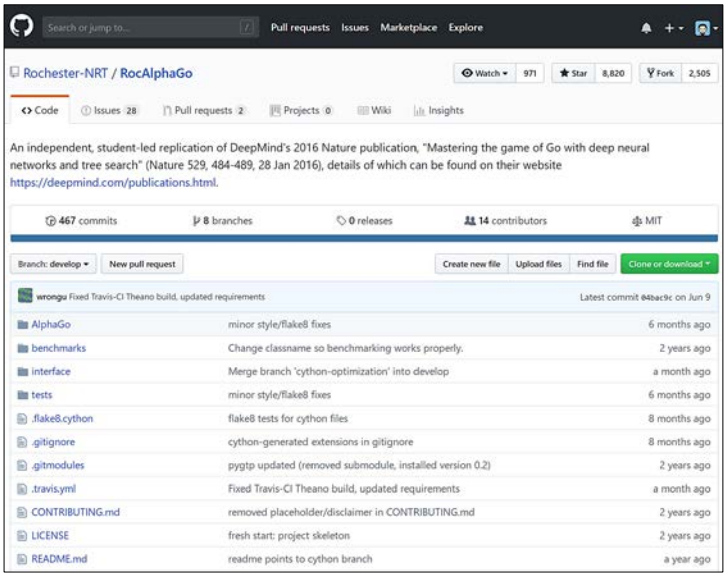


图 14-19　RocAlphaGo 在 GitHub 上的开源页面

14.2　AlphaGo Zero

如果说 AlphaGo 借助了大量围棋大师的棋谱才逐渐被培养成机器人围棋大师，那么 AlphaGo Zero 就纯粹是“自学成才”了。

2017 年 10 月 19 日，据媒体报道，谷歌旗下的人工智能团队 DeepMind 开发的 AlphaGo Zero 通过自学，以 100 比 0 的战绩击败了它的上一代——AlphaGo。

14.2.1　改进之处

关于 AlphaGo Zero 的实现，DeepMind 团队也将其写成了一篇论文 *Mastering the Game of Go without Human Knowledge*，发表在 2017 年的 *Nature* 上。从论文的名字就能看出，AlphaGo Zero 根本不需要学习围棋大师的棋谱。

AlphaGo Zero 的优点，大概可以总结为如下四个。

1. 残差网络

在 AlphaGo Zero 中，卷积神经网络部分采用了 Residual Network（残差网络）架构来做特征提取。Residual Network 在特征提取方面的优良能力，与过去的非“残差”的网络相比，有了很大的提升。

残差网络已经在第 6 章中介绍过了，它是一种带有 Shortcut 功能的 NIN 网络。在训练中，尝试了 20 个 Block、40 个 Block 等多个深度残差网络的版本。一个 Block 就是一个深度残差网络中

的 Shortcut 单元。

最终问世的 AlphaGo Zero，是一个经过 40 天训练的、由一个包含 40 个 Block 的深度残差网络实现的版本。

2. 左右互博

一个人的左手和右手下棋，是一件很痛苦（或者说奇怪）的事情吧？但是，AlphaGo Zero 就能这么做——只要把围棋规则告诉机器人，不需要参考任何人类棋手的棋谱，就能让机器人“自学成才”。这是一个自我冷启动过程。

AlphaGo Zero 的算法和工程实现比 AlphaGo 简单得多。AlphaGo 需要通过 SL Policy Network 来学习棋谱（做第一代的老师），通过 Rollout Network 快速落子（做 MCTS 搜索），通过 RL Policy Network 学习策略（进化），通过 Value Network 做估值计算（学习）。AlphaGo Zero 则没有这么复杂，如图 14-20 所示，在这个“左右互搏”的过程中，某个 Episode 的 s_1 状态到 s_T 状态，是通过 MCTS 的方法来估算落子概率的，最后的 s_T 状态会反馈一个是否胜利的标识 z，z 会通过回溯的方式不断向前传递，从而为每个状态的估值函数提供估值——和 AlphaGo 区别不大。

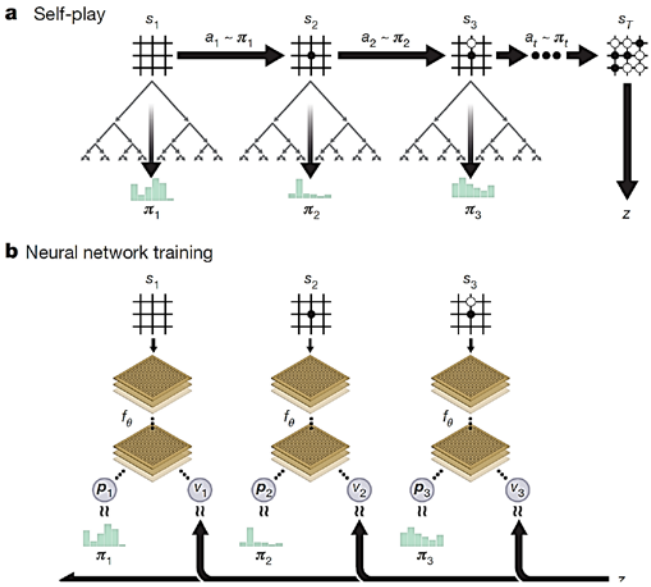


图 14-20　AlphaGo Zero 的训练方式

用于做特征提取的网络 f_θ，就是我们前面说的那个有 40 个 Block 的深度残差网络。这个网络在最后分叉了：一个是对落子的预测值 p_t，也就是 Policy Network；另一个是刚刚说到的估值函数 v_t，也就是 Value Network。

MCTS 的估值方式，如图 14-21 的 a、b 和 c 所示，与 AlphaGo 没有区别。

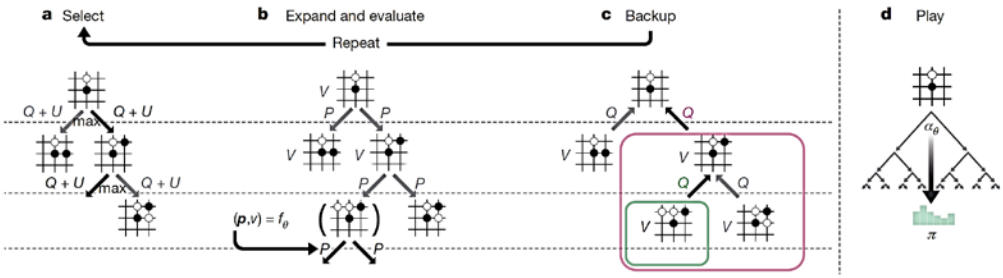
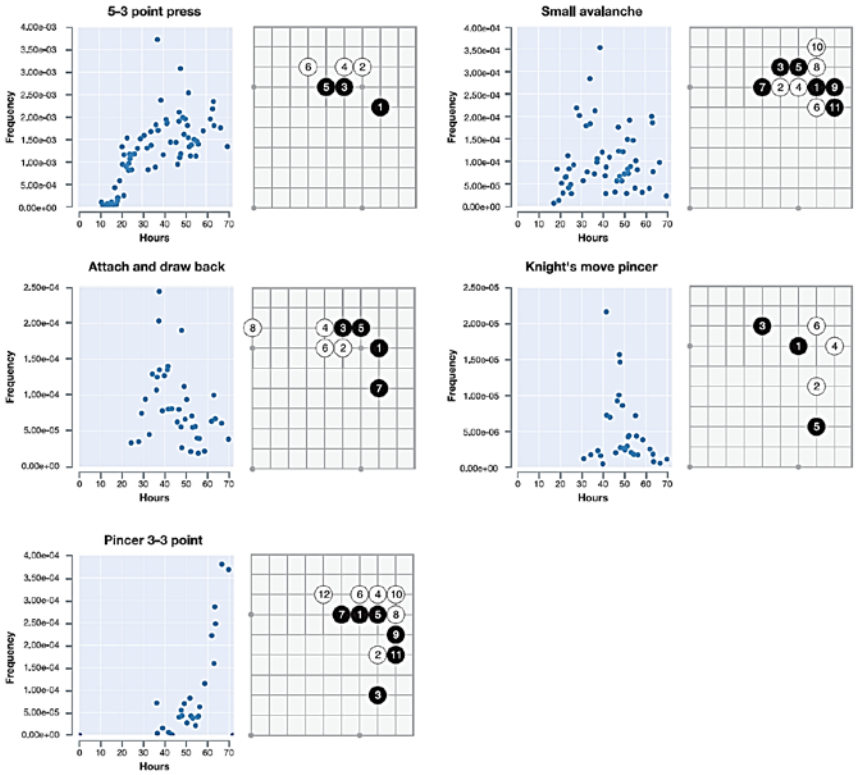


图 14-21　AlphaGo Zero 的 MCTS 估值和落子

3. 发现新定式

AlphaGo Zero 通过自己摸索，不仅总结出了定式等围棋战术，还发现了人类几千年来没有发现的新定式（如图 14-22 所示）。这就厉害了——机器人用了短短几十天的时间，就已经在技术水平和专业能力上全面超越人类，甚至可以给人类当老师了。



Extended Data Figure 2 | Frequency of occurence over time during training, for each *joseki* from Fig. 5a (corner sequences common in professional play that were discovered by AlphaGo Zero). The corresponding *joseki* are shown on the right.

图 14-22　AlphaGo Zero 发现的新定式

“定式”是一个围棋术语，是指千百年来约定俗成的落子套路，就好像——如果你这么走，我就那么走。这是一种心照不宣的、在双方都遵守的情况下收益还不错的情形。定式所形成的结果，就是在一个小范围内，双方的收益基本相当（这个收益主要指的是目或者子）。人类使用定式的根源，说句“白话”，就是利用经验来节省脑力。AlphaGo Zero 通过大量的左右互搏，发现了一些新的定式。这些定式是人类多年来没有发现的，或者说没有被主流围棋界接受并载入棋谱的。这些定式的形成在统计学上的依据是：这样落子会在最终导致双方的胜负概率相当。这是一个非常有趣的现象——AlphaGo Zero 发现人类已经发现的定式，是对人类智慧的量化证明，而发现新的定式，则是对人类智慧的突破。

4. 进步神速

AlphaGo Zero 从零开始自学，用了 40 天左右，就超越了 AlphaGo——AlphaGo 在此前几个月还曾以 60 比 0 的战绩战胜了众多人类围棋高手。

AlphaGo 各版本的对比，如图 14-23 所示。

版本	硬件	等级分	赛况
AlphaGo Fan	176个GPU、[4]分布式	3144[1]	5：0 对阵 樊麾
AlphaGo Lee	48个TPU、[4]分布式	3739[1]	4：1 对阵 李世石
AlphaGo Master	4个第二代TPU[4]、单机	4858[1]	60：0 对阵五位职业棋手 中国乌镇围棋峰会
AlphaGo Zero	4个第二代TPU[4]、单机	5185[1]	100：0 对阵AlphaGo Lee 89：11 对阵AlphaGo Master

图 14-23　AlphaGo 各版本的对比

为了区分不同的版本，DeepMind 给每个版本都起了名字。例如，战胜樊麾的那个版本叫 AlphaGo Fan，战胜李世石的那个版本叫 AlphaGo Lee，战胜 60 位成名大师的那个版本叫 AlphaGo Master（这几个版本，我们在前面笼统地称作“AlphaGo”）。

AlphaGo Zero 有一个 40 天版本和一个 34 小时版本。我觉得，对这些对比数据，了解一下就得了——毕竟训练的时间不一样，运行的硬件资源也不一样。不过，它们都厉害极了。40 天版本的 Elo 超过了 5100 分，应该是有史以来最高的分数了，它的战绩是：100 比 0 赢了 AlphaGo Lee、89 比 11 赢了 AlphaGo Master。

在 AlphaGo 和 AlphaGo Zero 的训练中，提到了 TPU 的概念（如图 14-24 所示）。

图 14-24　1 个 TPU 和 4 个带有散热器的 TPU 模块

TPU（Tensor Processing Unit）是谷歌自行研发的一种对深度学习进行了特殊优化的处理器。将“Tensor Processing Unit”强行翻译成中文，就是“张量处理器”（很少有人使用它的中文名）。与传统的使用 GPU 进行运算的方式不同，TPU 使用 8bit 的低精度计算功能，这样就在同样的硬件资源环境下，拥有了更强的计算能力、实现了更低的能耗，这对于高密度型的计算场景是有重要意义的。TPU 还使用了一种叫作“Systolic Array”的设计方式，优化了矩阵运算和卷积运算，大幅减少了 I/O 操作。除此之外，TPU 加大了板载缓存（类似于显卡的显存），从而进一步减少了 I/O 开销。

2017 年 5 月，谷歌发布了第二代 TPU 产品，并将其部署在谷歌的云平台上。据悉，这种 TPU 的浮点运算能力高达每秒 180 万亿次，但是主频仅有不到 1GHz（700MHz）。

第一代 TPU 和 GPU、CPU 的计算能力对比，如图 14-25 所示。CPU 的型号是 Intel Haswell E5-2699 v3，GPU 的型号是 Nvidia Tesla K80。“Total”包含主机的电能消耗。“Incremental”指的是增量，也就是一个 GPU 或者 TPU 的电能消耗。所以，单纯从能耗方面看，每消耗 1 瓦特电能，GPU 的计算能力只是 CPU 的 1.7 倍，而相同指标的 TPU 的计算能力是 GPU 的 25 倍。“Perf./Watt GM”和“Perf./Watt WM”分别指 Geometric Means（几何平均）和 Weighted Means（加权平均），具体的计算方法我没有查到。所以，这些数字做个参考就好，等以后有了更多的资料，咱们再在学术上进行探讨吧。

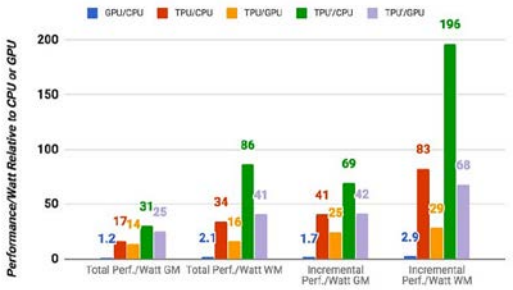


图 14-25　第一代 TPU、GPU、CPU 的计算能力对比 ①

14.2.2　成绩

AlphaGo Zero 的成绩高得让人惊掉下巴，除了以 100 比 0 完胜 AlphaGo Lee，还有一些量化的成绩，下面举个例子，如图 14-26 所示。

图 a 是 AlphaGo Zero 40 blocks、AlphaGo Master、AlphaGo Lee 三个版本的棋力对比。这里的“blocks”还是指深度残差网络的 Block 数量，数量越大，就说明网络越深，能够记忆和学习的内容也越丰富。从时间上看，AlphaGo Zero 只花了大约 3 天就超过了 AlphaGo Lee，大约 30 天就超

① 引自 https://arxiv.org/ftp/arxiv/papers/1704/1704.04760.pdf。

过了 AlphaGo Master。至于分数，只是一个数字，因为 AlphaGo Zero 已经是“逆天”的存在了。

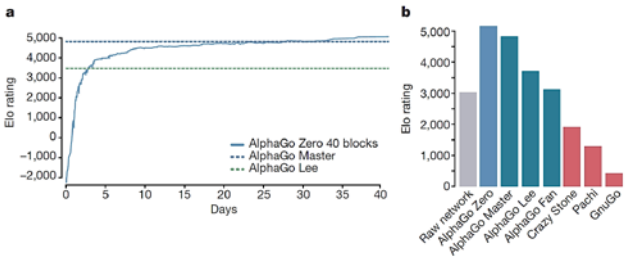


图 14-26　AlphaGo Zero 与 AlphaGo 的棋力对比

图 b 是与其他围棋软件的对比，差距还是很悬殊的。几个版本的 AlphaGo，水平都要比同时期的其他围棋软件高出一大截——一方面，DeepMind 团队的技术实力确实很强；另一方面，仍然是硬件上的优势。

14.2.3　开源项目

在 AlphaGo Zero 的论文发布不久，著名免费围棋软件 Leela 的作者就做了一个开源的实现并将其上传到 https://github.com/gcp/leela-zero，如图 14-27 所示。这个项目获得了 2500 多颗星，算是一个口碑比较好的开源项目了。

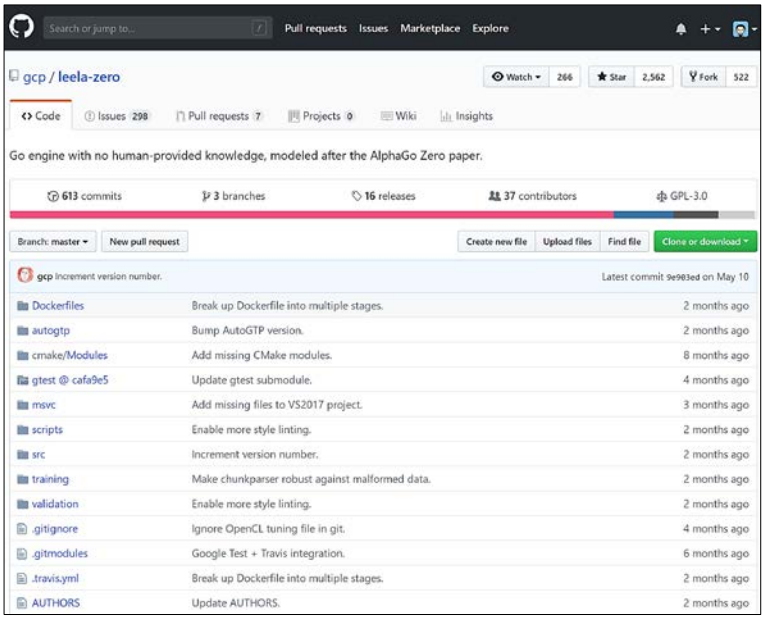


图 14-27　Leela 版 AlphaGo Zero 在 GitHub 上的开源页面

除此之外，可以找到一个由腾讯开源的项目 PhonenixGO，地址为 https://github.com/Tencent/PhoenixGo，如图 14-28 所示。这个项目也得到了将近 1900 颗星。

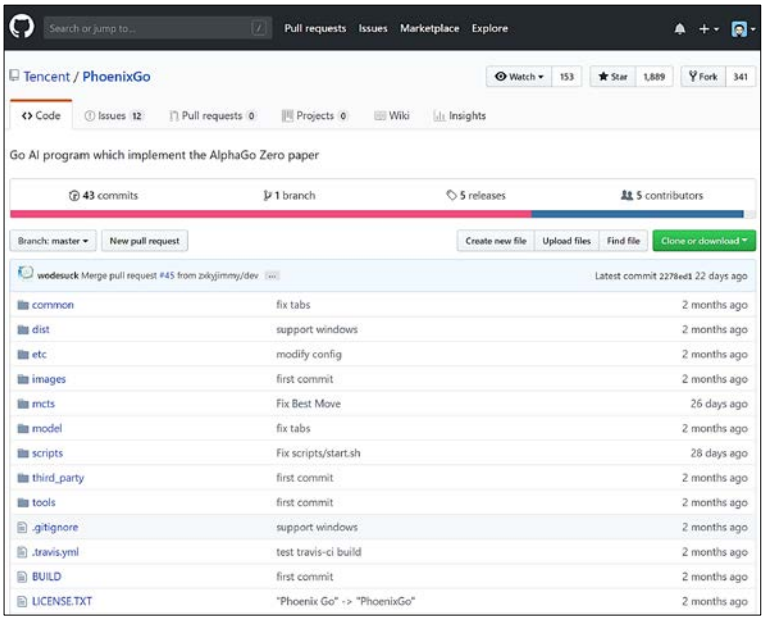


图 14-28　PhoenixGo 在 GitHub 上的开源页面

这两个项目的代码都是用 C++ 编写的。不熟悉 C++ 的读者朋友读起来可能会有点吃力。

不过，我没有找到用 PyTorch 实现的 AlphaGo 或者 AlphaGo Zero 的版本。我想，这可能有两个原因。一个原因是 PyTorch 的生态还不完善，社区的热度也不理想，不足以吸引技术高手用它来编写一个如此复杂的机器人程序。另一个原因可能与性能有关，即完全使用 C++ 实现的模型，不管是正向传播，还是反向传播，速度都要比基于 Python 的方法快一些（对这一部分的研究尚处空白期，是我们可以选择的一个方向）。

关于要不要自己用 PyTorch 实现一个 AlphaGo 模型，我说点个人意见吧。从负责任的角度来讲，我觉得还是不要轻易尝试。

第一，训练这样一个机器人会消耗大量的资源，这意味着消耗时间和金钱——干脆一点说，就是意味着消耗金钱。无论是公司还是个人，金钱这种资源都是有限的，应该用来做更有价值、更容易产生成果和实际效益的事情。

第二，AlphaGo 所针对的围棋 AI 领域，已经是一个被世界顶级人工智能研究团队攻克的命题了。如果想在这个领域做得更好，那么，对单纯以“秀肌肉”为目的的研究，我个人是不太提倡的。哪怕是对非营利研究机构而言，我也认为，应该去挑战一些目前研究相对较少的领域，这样才容易填补空白，从而实现“弯道超车”。当然，与一个投入巨大而结果极其不确定的领域相比，

在那些研究相对较少的领域，也比较容易出成绩。

所以，AlphaGo 也好，AlphaGo Zero 也罢，我们借鉴其中的思路与方法就好了，不必非得去仿制一个。对于实在技痒且对自己的能力非常有自信的读者朋友，我建议，可以弄一个 9×9 或者更小的棋盘试试水——因为技痒而自己掏腰包买 100 块显卡，确实没必要。

14.3　试验场大观

关于深度学习的研究，别的优势咱找不到，但是其本身的立意，足够保证它的实验对象有无穷多种。谁让深度学习的创立目的就是像人一样解决所有的贯序决策问题呢？从根本上，我们就期望它是一种万能的方式。

为了给热衷研究人工智能的“芸芸众生”找到合适的“玩具”，不少爱好者做了很多的工作。

14.3.1　StarCraft Ⅱ

《星际争霸Ⅱ》（StarCraft II）是《星际争霸》（StarCraft）系列游戏的第二个较大的版本。第一代《星际争霸》是暴雪（Blizzard）公司于 1998 年 3 月发行的。当时，《星际争霸》在个人 PC 游戏市场有相当高的占有率，几乎是每个中学男生的标配游戏。

《星际争霸Ⅱ》是在 2010 年 7 月发行的，沿用了人族、神族、虫族在星际间征战的故事体系，如图 14-29 所示。不过，由于其用户基础比较好，现在还有很多三四十岁的玩家对这个游戏爱不释手。

图 14-29　《星际争霸Ⅱ》游戏场景

作为一款经典的即时战略类游戏，《星际争霸》无疑是成功的——无论是在故事设置、游戏互动性、角色实力平衡性方面，还是在美工方面。暴雪不愧是世界一流的游戏公司。

1. 环境信息

为了满足广大人工智能爱好者的需求，《星际争霸Ⅱ》在其官方网页上披露了相关的 API 信息，列举如下。

- StarCraft II API Library：https://github.com/Blizzard/s2client-api，如图 14-30 所示。
- StarCraft II API Protocol：https://github.com/Blizzard/s2client-proto，如图 14-31 所示。
- Python Protocol Binding Library：https://pypi.org/project/s2clientprotocol/。
- PySC2 (DeepMind's toolset)：https://github.com/deepmind/pysc2，如图 14-32 所示。

请参考附录 A.7.5 节安装《星际争霸Ⅱ》，然后进行下面的操作。

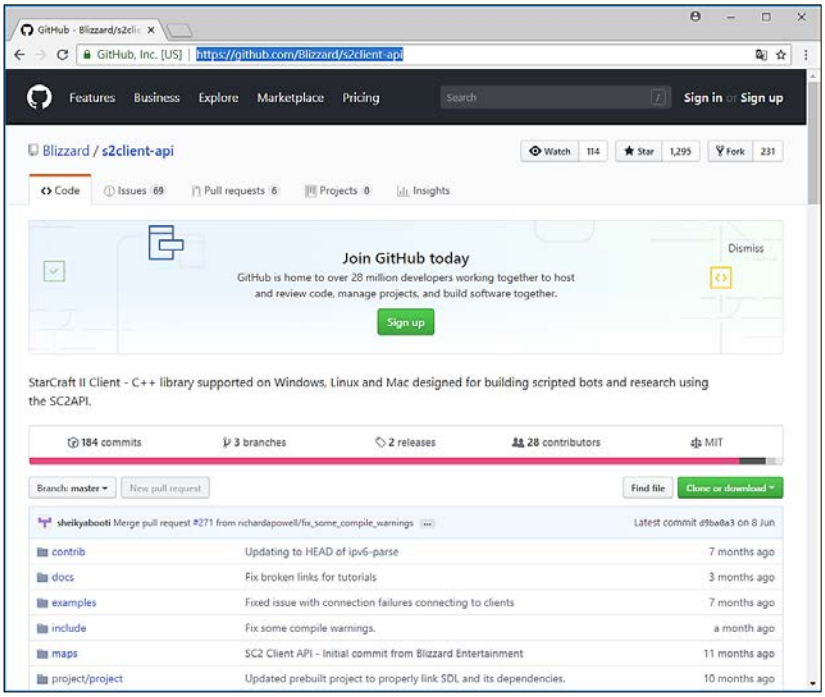


图 14-30　StarCraft II API Library 在 GitHub 上的主页

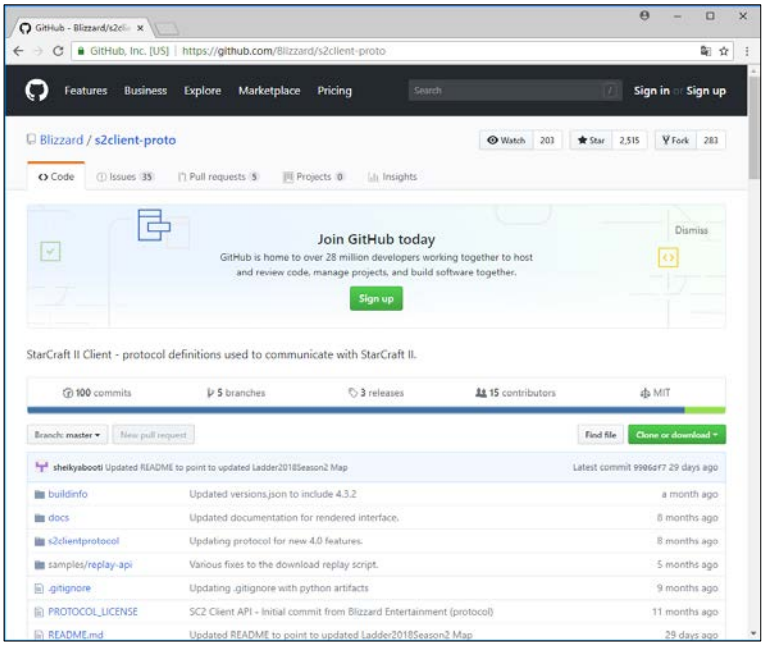

GitHub - Blizzard/s2cli
GitHub, Inc. [US] | https://github.com/Blizzard/s2client-proto
Features Business Explore Marketplace Pricing
Search
Sign in or Sign up
Blizzard / s2client-proto
Watch 203 Star 2,515 Fork 283
Code Issues 35 Pull requests 5 Projects 0 Insights
Join GitHub today
GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.
Sign up
Dismiss
StarCraft II Client - protocol definitions used to communicate with StarCraft II.
100 commits 5 branches 3 releases 15 contributors MIT
Branch: master New pull request
Find file Clone or download
sheikyabooti Updated README to point to updated Ladder2018Season2 Map
Latest commit 9906df7 29 days ago
buildinfo Updated versions.json to include 4.3.2 a month ago
docs Updated documentation for rendered interface. 8 months ago
s2clientprotocol Updating protocol for new 4.0 features. 8 months ago
samples/replay-api Various fixes to the download replay script. 5 months ago
.gitignore Updating .gitignore with python artifacts 9 months ago
PROTOCOL_LICENSE SC2 Client API - Initial commit from Blizzard Entertainment (protocol) 11 months ago
README.md Updated README to point to updated Ladder2018Season2 Map 29 days ago

图 14-31　StarCraft II API Protocol 在 GitHub 上的主页

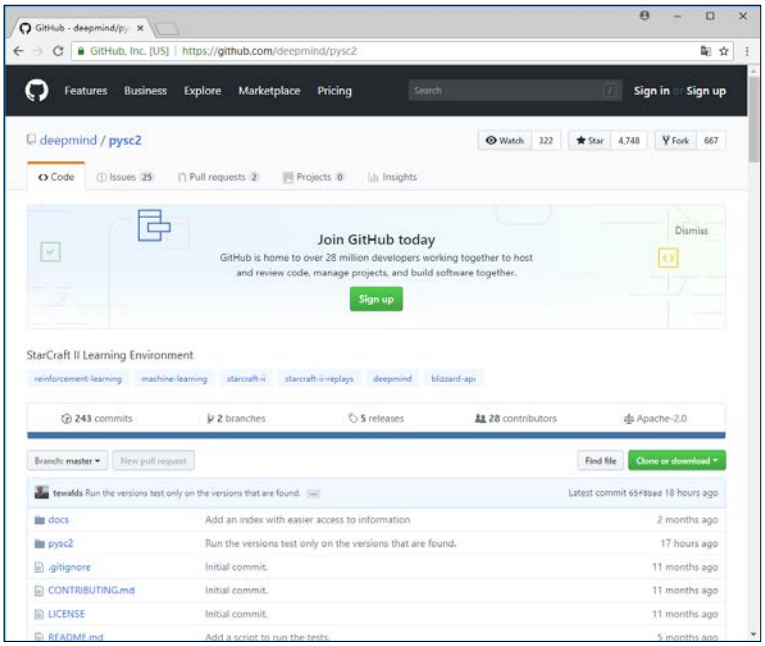

GitHub - deepmind/py
GitHub, Inc. [US] | https://github.com/deepmind/pysc2
Features Business Explore Marketplace Pricing
Search
Sign in or Sign up
deepmind / pysc2
Watch 322 Star 4,748 Fork 667
Code Issues 25 Pull requests 2 Projects 0 Insights
Join GitHub today
GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.
Sign up
Dismiss
StarCraft II Learning Environment
reinforcement-learning machine-learning starcraft-ii starcraft-ii-replays deepmind blizzard-api
243 commits 2 branches 5 releases 28 contributors Apache-2.0
Branch: master New pull request
Find file Clone or download
tewalds Run the versions test only on the versions that are found.
Latest commit 65f80ad 18 hours ago
docs Add an index with easier access to information 2 months ago
pysc2 Run the versions test only on the versions that are found. 17 hours ago
.gitignore Initial commit. 11 months ago
CONTRIBUTING.md Initial commit. 11 months ago
LICENSE Initial commit. 11 months ago
README.md Add a script to run the tests. 5 months ago

图 14-32　《星际争霸Ⅱ》的游戏接口

安装《星际争霸Ⅱ》的游戏引擎和 DeepMind 提供的 PySC2 之后，你就可以以普通人类玩家的身份让一个游戏机器人来玩这个游戏了（训练）。

最简单的方式是，创建一个 Agent，编辑 simple_agent.py 文件，具体如下。

```
from pysc2.agents import base_agent
from pysc2.lib import actions

class SimpleAgent(base_agent.BaseAgent):
    def step(self, obs):
        super(SimpleAgent, self).step(obs)

        return actions.FunctionCall(actions.FUNCTIONS.no_op.id, [])
```

然后，启动这个 Agent，代码如下。

```
python -m pysc2.bin.agent \
--map Simple64 \
--agent simple_agent.SimpleAgent \
```

启动之后，你就会看到一群“农民”在玩儿命采矿，而且只采水晶矿。由于游戏默认是加速训练的，过不了 1 分钟，就会有敌人的“小兵”过来掏我们的“老窝”了。中间那个会逐渐扩大的深色区域就是失去的“血量”，只要“老窝”被攻陷，游戏就结束了，如图 14-33 所示。

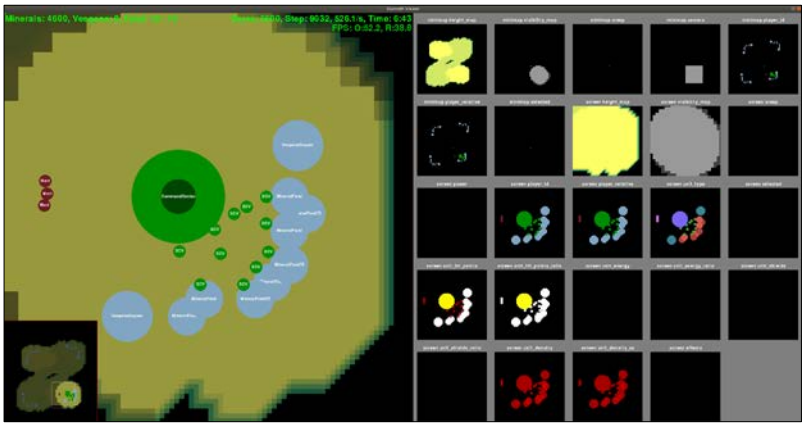


图 14-33　启动《星际争霸Ⅱ》的 Agent

在《星际争霸Ⅱ》中，经常会出现两个或者多个玩家混战的情况，目的极为单纯，就是把其他玩家的“手下”都“杀”光，只不过过程略显复杂。要想训练一个完整的游戏 Agent AI，我们必须知道输入的 State 是什么，动作空间 Action 都有什么，以及每一步是不是有有效的 Reward 可以获得。

在刚刚我们做实验的这幅小地图（Simple64）上，只有一个玩家和你“征战”，你们俩一个在

左上角，一个在右下角。

如果你在程序里使用

```
dir(obs)
```

函数，就可以查看 obs 信息，如图 14-34 所示。

```
E0714 20:15:50.213249 139811611002688 renderer_human.py:205] Failed to get the resolution from xrandr.
I0714 20:15:50.328222 139811611002688 sc2_env.py:306] Environment is ready on map: Simple64
I0714 20:15:50.328772 139811611002688 sc2_env.py:433] Starting episode: 1
   0/no_op                                          ()
   1/move_camera                                    (1/minimap [64, 64])
   2/select_point                                   (6/select_point_act [4]; 0/screen [84, 84])
   3/select_rect                                    (7/select_add [2]; 0/screen [84, 84]; 2/screen2 [84, 84])
   4/select_control_group                           (4/control_group_act [5]; 5/control_group_id [10])
   5/select_unit                                    (8/select_unit_act [4]; 9/select_unit_id [500])
264/Harvest_Gather_screen                           (3/queued [2]; 0/screen [84, 84])
 12/Attack_screen                                   (3/queued [2]; 0/screen [84, 84])
 13/Attack_minimap                                  (3/queued [2]; 1/minimap [64, 64])
269/Harvest_Return_quick                            (3/queued [2])
274/HoldPosition_quick                              (3/queued [2])
 51/Build_EvolutionChamber_screen                   (3/queued [2]; 0/screen [84, 84])
 52/Build_Extractor_screen                          (3/queued [2]; 0/screen [84, 84])
451/Smart_screen                                    (3/queued [2]; 0/screen [84, 84])
452/Smart_minimap                                   (3/queued [2]; 1/minimap [64, 64])
453/Stop_quick                                      (3/queued [2])
331/Move_screen                                     (3/queued [2]; 0/screen [84, 84])
332/Move_minimap                                    (3/queued [2]; 1/minimap [64, 64])
333/Patrol_screen                                   (3/queued [2]; 0/screen [84, 84])
334/Patrol_minimap                                  (3/queued [2]; 1/minimap [64, 64])
230/Effect_Spray_screen                             (3/queued [2]; 0/screen [84, 84])
  6/select_idle_worker                              (10/select_worker [4])
^CTook 10.738 seconds for 95 steps: 8.847 fps
I0714 20:16:01.090974 139811611002688 sc2_env.py:531] Environment Close
```

图 14-34　obs 信息

如果执行

```
returns = super(SimpleAgent, self).step(obs)
dir(returns)
```

会返回如图 14-35 所示的内容。

```
I0714 20:21:07.080443 140098970806080 sc2_env.py:433] Starting episode: 1
   0/no_op                                          ()
   1/move_camera                                    (1/minimap [64, 64])
   2/select_point                                   (6/select_point_act [4]; 0/screen [84, 84])
   3/select_rect                                    (7/select_add [2]; 0/screen [84, 84]; 2/screen2 [84, 84])
   4/select_control_group                           (4/control_group_act [5]; 5/control_group_id [10])
Took 6.252 seconds for 56 steps: 8.957 fps
```

图 14-35　返回的内容

这个 obs 信息，是用于描述 Observation 的环境信息——这肯定是没跑儿了。如果你想全面了解环境能够以哪些值作为输入，还是要查询说明文档。在这个页面上可以找到比较全面的介绍：https://github.com/deepmind/pysc2/blob/master/docs/environment.md。

对这些输入的信息，我建议不要一股脑地当作输入的 State，而要有所取舍，即取一个子集。

《星际争霸Ⅱ》的 Action 还是比较特别的。与 Atari 游戏里的 Agent 不同，《星际争霸Ⅱ》里面的这些单位的数量和种类是不同的。作为一个 Agent AI，不一定在任何时候都有相同的 Action 选择空间。例如，这一刻你有 3 个“农民”，可以命令他们都去采矿；下一刻你的“农民”被敌人

全数歼灭，你就无法将命令下达给“农民”了，Action 的可选范围也肯定会发生变化。

如果要查阅所有的 Action 信息列表，可以使用如下命令。

```
python -m pysc2.bin.valid_actions
```

我截取了最后几行输出内容，如图 14-36 所示。

```
 530/Effect_Repair_RepairDrone_screen                (3/queued [2]; 0/screen [84, 84])
 531/Effect_Repair_RepairDrone_autocast              ()
 532/Effect_RepairDrone_screen                       (3/queued [2]; 0/screen [84, 84])
 533/Effect_Restore_screen                           (3/queued [2]; 0/screen [84, 84])
 534/Effect_Restore_autocast                         ()
 535/Morph_ObserverMode_quick                        (3/queued [2])
 536/Morph_OverseerMode_quick                        (3/queued [2])
 537/Morph_OversightMode_quick                       (3/queued [2])
 538/Morph_SurveillanceMode_quick                    (3/queued [2])
 539/Research_AdaptiveTalons_quick                   (3/queued [2])
 540/Research_CycloneRapidFireLaunchers_quick        (3/queued [2])
Total base actions: 541
Total possible actions (flattened): 102065741
```

图 14-36　Action 空间

可选的动作空间有 541 个，也就是说，有 541 个不同的动作类别。每个动作类别还有不同的取值，例如攻击 Attack_minimap（攻击小地图上的某个点），这个动作有 64 × 64 的空间，也就是有 4096 个不同的取值。这样算下来，一共有 102065741 个不同的动作取值——是的，超过 1 亿个……如果想返回某个动作，在 `step()`函数的尾部返回就可以了（在前面的程序里就有相应的命令）。其实，一旦这个结构明确了，在模式上，感觉就和 Gym 一样了。那么，剩下的就是自己编写一个模型来进行训练了。

需要说明一点，《星际争霸Ⅱ》和很多复杂的即时战略类游戏一样，都没有提供合适的奖励值体系。就连围棋游戏中的奖励值体系，也不是由围棋游戏本身提供的。从本质上看，这一点和 Gym 中的 Atari 游戏是相同的。为什么我们的感觉不一样呢？Atari 的评价体系比较简单，基本上就是靠得分，得分高的自然可以获得胜利（或者容易取得胜利）——这个评价体系非常单纯。在这种情况下，将每一步的反馈得分作为 Reward 值，然后做时间差分，问题是不大的。可《星际争霸Ⅱ》及围棋这种游戏就不同了。在它们的环境中，确实有一些关于击杀对方单位（游戏角色或棋子）的描述，也有关于被对方击杀的描述，而这些信息很难被描述为奖励值。我的观点是，奖励值应该是一个综合描述动作对于目标达成的重要程度的值，所以，它在这种复杂的环境中不方便也不应该以单纯的方式提供。奖励值在很大程度上反映了定义者的“价值观”，如果定义者的“价值观”本身有问题，就会在很大程度上误导机器人。

所以，我们可以通过 AlphaGo 的例子看出，Reward 在训练的过程中其实不是从每个 Step 中获得的，而是通过 MCTS 算法，不断试探从这条路往下走有多大的概率获胜，并通过概率评价来不断对 $V(s)$ 的真实值进行逼近的。只要 $V(s)$ 的值比较准确，$Q(s,a)$ 也会变得准确，于是，$R(s,a)$ 是否存在就变得不重要了。我觉得，对这个观点，大家应该有基本的了解，不要一看没有奖励值就觉得很绝望。

当然，在没有奖励值的情况下（也就是在评价体系比较复杂的情况下），训练是非常艰难的。例如《星际争霸》这样的游戏，如果你真的老老实实用 MCTS 的方法去搜索，其复杂程度比围棋高了不知道多少倍，而且……只能跟大家道个歉，以我的能力，暂时没办法准确估算这个空间的大小——这绝对不是比 AlphaGo 的复杂程度高几个数量级那么简单的。如果想进行训练，只能借助其他技巧。

2. 业界状况

2016 年 11 月，DeepMind 团队宣布与暴雪合作，将针对《星际争霸Ⅱ》展开一系列的研究和实验。2017 年 8 月 7 日，Facebook 的 AI 团队发布了《星际争霸：母巢之战》的游戏数据集。2017 年 8 月 10 日，DeepMind 团队和暴雪联合发布了《星际争霸Ⅱ》的游戏数据集及 AI 训练平台 SC2LE。面对这一绝对的世界级难题，两个世界上的著名 AI 组织都表现出了极大的热情与诚意。

顺便说一下：这里可不是我的笔误，Facebook 不是一家单纯地经营网络社交媒体的公司，而是一家顶尖的科技公司，本书涉及的深度学习框架 PyTorch 就是它的力作（在深度学习界颇有影响力的 Caffe 和 Caffe2 框架也是）。

不过，在一次完整的 1 对 1 战斗中，AI 的表现还是非常弱的，可以说，如果没有理论层面的突破，没有一些 Bug 级技巧的应用，《星际争霸Ⅱ》的 AI 的训练仍旧艰难非常。

大家也别泄气，由于社群的逐渐壮大，以及研究者的不断加入，虽说在战略层面取得进步是比较困难的，但在战术层面的进步还是有目共睹的。这里说的战术层面，就是指在一次战斗的局部设置较多的限制条件。在这些条件的限制下，训练的难度将会降低，甚至不需要使用很多的计算资源就能得到比较好的结果。

例如，在 10 对 10 的机枪兵战斗场景中，AI 能够学会在战斗的过程中集中优势火力、消灭对方的有生力量，所谓“伤其十指不如断其一指”。当然，对决的双方是由强化学习（或者类似的可进化的方法）训练的 AI 和游戏中自带的脚本型 AI。

一些视频可以供大家参考。这些视频大都是由非营利机构和人工智能发烧友们发布的，推荐大家访问 YouTube 观看。例如，在 https://www.youtube.com/watch?v=5dBEXAs8mOI 这段视频里，都是“以弱胜强”的片段，内容是一些战斗单位较少、较弱但战术素养较高的 AI 挑战数量多于自己的普通脚本式 AI 的战斗。在视频的一开始，5 个 Stalker（追猎者）遭遇 15 个 Marrien（机枪兵），如图 14-37 所示。这些 Stalker 会采用“各自为战”“打一枪换一个地方”“敌进我退，敌驻我扰，敌疲我打”等战术来机动性地消耗对方——简直是一群熟读兵法的机器人，而且它们的兵法是自己摸索、总结出来的。

https://www.youtube.com/watch?v=IKVFZ28ybQs 这段视频的场面看上去更激烈一些。这是一个 100 个 Zergling（小狗）冲击 20 个 Siege Tank（攻城坦克）的阵地的挑战项目，如图 14-38 所示。在这样的游戏场景中，即便是微操作能力比较强的人类玩家，也只能“干掉”几辆坦克，肯定无法“消灭”整个坦克群。在视频中有一段文字说明：“100 zerglings bum rushing into 20 siege

tanks results in all zerglings dying without killing more than 2 tanks :(”。意思是：“100 只小狗闯入 20 辆攻城坦克中间，结果所有小狗都被击杀而无法击杀 2 辆以上的坦克”。在这段文字的结尾有一个嘴角下垂的表情符号，用来表示沮丧的心情。玩过《星际争霸Ⅱ》的读者朋友都知道，攻城坦克的火炮是有溅射效果的，在开炮时能击杀附近的目标，而这对玩家来说是非常“恐怖”的——无论由谁来操作，都很难避开几十辆坦克同时开火而形成的火力网。

图 14-37　5 个追猎者对战 15 个机枪兵

图 14-38　100 只小狗冲击 20 辆攻城坦克的阵地

高级 AI 出场后，出现了一个非常有趣的现象。如图 14-39 所示，视频通过慢动作告诉我们，行进中的小狗队伍突然闪出了一个空档，不为别的，就为了躲避一枚即将落地爆炸的炮弹。紧接着，炮弹爆炸了，但只对小狗队伍造成了很小的伤害，小狗队伍继续向攻城坦克群冲锋。这样的情况出现了若干次，每次都只有个别的小狗被炸死，其余的小狗依然向坦克群冲锋。

最后，这 100 只小狗不仅把 20 辆攻城坦克全部“消灭”了，还有几十只存活下来，穿过阵地，到达了指定位置，如图 14-40 所示。

图 14-39　高级 AI 操控小狗躲避炮弹

图 14-40　高级 AI 操控小狗冲过攻城坦克阵地

没有公开的信息披露具体的实现方法或者训练过程。不过，不管训练是如何进行的，我个人认为，这群小狗真是棒棒的。想想看：用一个脚本控制 100 只小狗躲避炮火的难度有多大？是用行为树脚本实现它比较方便，还是用强化学习的方法实现它比较方便？我觉得，后者起码是一个可以依靠 AI 来发现更快、更好方法的过程，而前者的实现完全依赖个人的编程能力，因此，应该优先考虑用后者来实现。你说对不对？

3. 开源项目

GitHub 上关于《星际争霸Ⅱ》的开源项目少得可怜。

https://github.com/SoyGema/Startcraft_pysc2_minigames 是一个得到了 50 多颗星的项目（如图 14-41 所示）——别觉得不满意，这是用“Starcraft2”和“AI”关键词搜索出来的星最多的项目了。

在这个项目里也有用 DQN 编写的机器人，它是使用 Keras 实现的（也就是说，如果要做相同的实验，还需要安装 TensorFlow）。大家可以把它下载下来玩玩看。如果有兴趣，也可以自己写一个网络来试试看。

不过，我觉得，对于完整的游戏，大型研究机构可以用它来磨自己的“屠龙刀”，而个人或者小型机构是训练不出结果的——如果要训练，找一个场景较小、任务较单一的游戏尝试一下就可以了。不管怎样，这个任务都是非常艰深的！

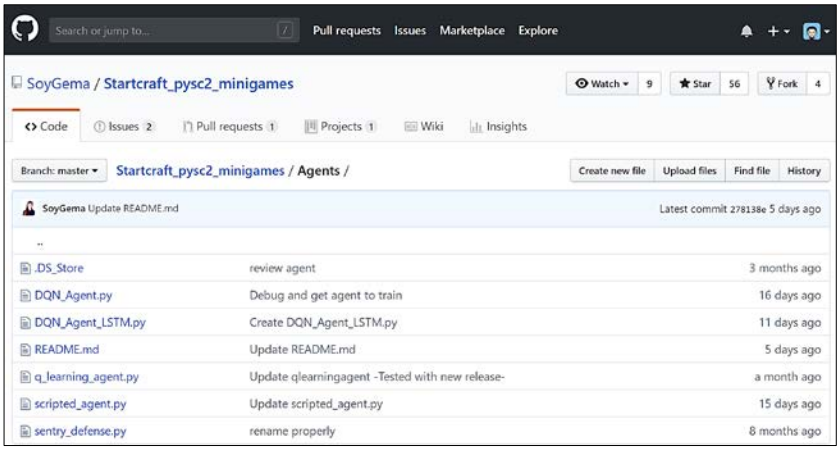


图 14-41 Startcraft_pysc2_minigames 的 GitHub 主页

14.3.2 VizDoom

20 世纪 90 年代，在“古典 PC”上有一款运行在 DOS 系统下的叫作“Doom”的游戏，中文名称叫作《毁灭公爵》，“80 后”们可能不陌生。我和我的小伙伴们在当年也玩过这款游戏。不过，由于当时的渲染算法等问题，游戏的画面容易让人晕眩，我自己也是玩一会儿就感觉眼花，我的一位同学甚至因此呕吐……

我们还是直接来看关于 Doom 的项目吧。该项目的主页为 http://vizdoom.cs.put.edu.pl/。这个网站隶属波兰的波兹南理工大学（Poznan University of Technology，PUT）。这是一所创立于 1955 年的著名的理工大学——可以理解为波兰的“211”或者“985”院校。

打开网站，你会看到一个外表有点简陋的页面（如图 14-42 所示）——比 MNIST 的主页强一点，但很有限。

说实在的，VizDoom 的显示效果，比我小时候玩 Doom 那会儿强多了（如图 14-43 所示）。我不清楚，为了将这个“古老”的游戏从 DOS 环境迁移到 Windows 10 和 Linux 环境中，究竟做了多少优化，至少我试着玩了一会儿，没有感到任何不适。这个游戏的内容相对比较单一，就是在一个密闭的房间中，以第一人称视角，使用手中的武器对敌进行击杀——大都是一些分不清种类的怪物。

VizDoom 的安装还是比较简单的，至少没有 Gym 里的 MuJoCo 那么复杂。具体安装步骤请参考附录 A.7.6 节。

训练时输入的 State 不像 Gym 中那样。它被封装成了 `env.step()`的一个返回值，是通过多个类来获取相关环境数据的。

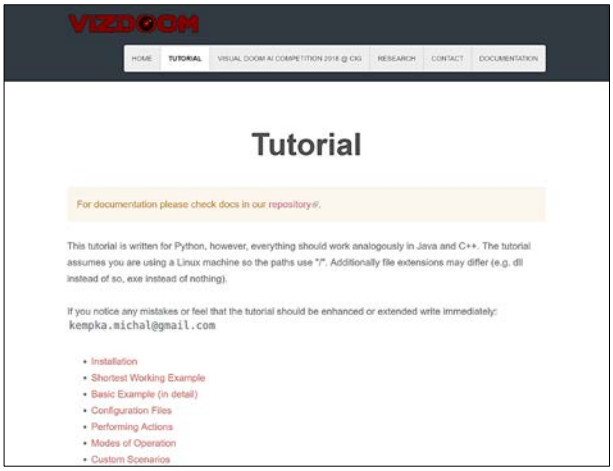

VIZDOOM
HOME
TUTORIAL
VISUAL DOOM AI COMPETITION 2018 @ CIG
RESEARCH
CONTACT
DOCUMENTATION
Tutorial
For documentation please check docs in our repository.
This tutorial is written for Python, however, everything should work analogously in Java and C++. The tutorial assumes you are using a Linux machine so the paths use "/". Additionally file extensions may differ (e.g. dll instead of so, exe instead of nothing).
If you notice any mistakes or feel that the tutorial should be enhanced or extended write immediately:
kempka.michal@gmail.com
Installation
Shortest Working Example
Basic Example (in detail)
Configuration Files
Performing Actions
Modes of Operation
Custom Scenarios

图 14-42　VizDoom Tutorial

图 14-43　VizDoom

例如，你可以简单地通过如下语句获取环境信息。

```
state = game.get_state()
       # Which consists of:
       n          = state.number
       vars       = state.game_variables
       screen_buf  = state.screen_buffer
       depth_buf   = state.depth_buffer
       labels_buf  = state.labels_buffer
       automap_buf = state.automap_buffer
       labels     = state.labels
```

在 basic.py 这个程序中，如果你找到 `state.screen_buffer` 这一行，就会发现，把它打印出来，会得到一个 $480 \times 640 \times 3$ 的张量（其实就是一个 640×480 的图像信息描述）。这个图像信息可以在游戏开始之前由

```
game.set_screen_resolution(ScreenResolution.RES_640X480)
```

这样的语句来设置。

除了像素级别的输入信息，游戏还提供了其他接口来帮助我们获取一些有用的数据：

```
AMMO2 = game.get_game_variable(GameVariable.AMMO2)
print('AMMO:',AMMO2)
HEALTH = game.get_game_variable(GameVariable.HEALTH)
print('HEALTH:', HEALTH)
```

通过这样的语句，很容易就可以获得当前玩家的子弹存量及血量信息，而这些都可以作为 State 的重要游戏数据使用。

动作输出的定义可能和你想象得略有不同，它是通过这样的方式定义的：

```
…
game.add_available_button(Button.MOVE_LEFT)
game.add_available_button(Button.MOVE_RIGHT)
game.add_available_button(Button.ATTACK)
…
actions = [[True, False, False], [False, True, False], [False, False, True]]
…
r = game.make_action(choice(actions))
```

换句话说，在用 `game.add_avaiable_button` 定义按键动作以后，在后面的 `actions` 动作空间中就可以任意进行组合了。用 `game.make_action()`实现 Action 输入，就相当于输入了一个诸如 `[True, False, False]` 的向量（表示按下了“左”按钮，而没有按下“右”和“上”按钮）。理解起来非常简单，关键是可以组合。例如，`[True, False, True]` 表示将“左”和“射击”按钮一起按下。

与 Gym 类似，在完成 `game.make_action` 操作后会得到一个 Reward。在上面的示例中，就是用 `r` 来接受赋值。

如果一个动作没有击杀任何敌人，就会得到负的 Reward；如果一个动作击杀了敌人，就会得到正的 Reward。剩下的工作就很简单了——可以考虑使用 DQN 或者我们在前面接触过的任何一种网络进行训练。

不要觉得训练这么简单的游戏没有意义，人家可是有正式锦标赛的。在 VizDoom 网站的明显位置有专门的宣传链接（如图 14-44 所示）。

2018 年的锦标赛有两个比赛项目，我们从网站上援引一下：

“The participants of the Visual Doom AI competition are supposed to submit an agent (in Python or Lua) that plays Doom using mainly visual input. Our ViZDoom framework gives a real-time access to the screen as the only information the agent can base its decision on. This year, there are two completely different tracks.

“Track 1 challenges agents to beat single player levels as fast as possible. Levels vary in difficulty

so the entry threshold is low - you do not need sophisticated knowledge to start and can learn on the run!

“Track 2 is a full on Doom deathmatch (like in previous years) on unknown maps. Agents will compete in multiplayer games and the best frag collector will emerge victorious.”

图 14-44　VizDoom Competition 主页

翻译成中文，大意如下。

“参加 Visual Doom AI 锦标赛需要提交一个训练好的用 Python 或者 Lua 语言实现的 Agent，并主要使用可视化的输入（屏幕输入信息）。平台的框架将提供一个实时连接到屏幕的接口。2018 年有两个不同的比赛项目。

“项目一：挑战者要在单人模式下尽快通关。比赛将分为多个难度等级，所以参加的门槛是相当低的。参与者无须具备高深的知识，并且可以边参赛、边学习。

“项目二：这是一个在未知地图上进行的完整的 Doom 竞赛，形式与 2017 年类似。在多人比赛的场景中，击杀敌人最多的参赛者为胜利者。”

虽说 Doom 这种纯属血腥杀戮的游戏内容我个人并不喜欢，但这种鼓励边参赛、边学习且策略相对单一的比赛规则，我还是比较喜欢的，因为它适合更多的参与者。也只有参与者众多的项目，才会在未来获得更多、更有趣的玩法和更广阔的生态环境。

最后，建议有精力的读者朋友都可以试试看，这个项目还是挺有趣的，而且与前面几个案例相比更容易得到结果。

14.3.3　Universe

在本章的前两节中，我们一起了解了两个经典的项目被“改装”后的训练环境，相信这些已经够大家玩一阵子了。在这一节中，我们一起来认识一个大项目——Universe（https://github.com/openai/universe）。

“Universe”，本意是“宇宙”——一个恢宏的名字。Universe 可以让一个 Agent AI 像人一样使用计算机：观看屏幕像素并操作虚拟键盘和鼠标。项目的设计者期望 Universe 能够在各种基于 PC 的任务上训练 AI，也就是说，通过 Universe 让单个 Agent 在任何人类可以使用计算机完成的任务中进行训练。借助 Universe，任何程序都可以被转换成一个类似 Gym 的环境，如图 14-45 所示。Universe 厉害的地方在于，它可以通过自动启动程序在一个 VNC[①]远程桌面上工作，所以，它不需要特别访问任何程序内部、源码或者机器人 API。简单地说，就是用一个虚拟摄像头对着计算机的屏幕，通过不断给机器人发放奖励值，最终让它向着我们设定的目标进化。因此，理论上 Universe 根本不在乎机器人正在做什么。

图 14-45　Universe 正在玩游戏

从立意和形式上看，Universe 跟 SerpentAI 非常像。其实，先问世的是 Universe，SerpentAI 纯粹是出于好玩而设计的……在我看来，更重要的是 Universe 的各种奇怪的设置和限制。我不太喜欢使用 Docker、VNC 这样的复杂部件——也确实可以不用。

Universe 的目的是训练出适应性更广的 Agent AI，让它把一切都当成游戏来玩。这个假设不可谓不大胆，甚至可以说是极有想象力的。当然，我们非常清楚，屏幕像素的输入都有数百万个维度，如果想训练这种大维度的输入，不采取点技术手段恐怕是不行的（例如，SerpentAI 在训练前进行了压缩）。即便如此，要想训练出成果，仍然需要大量的硬件资源。

如果你想在 Universe 和 SperentAI 之间做个选择，我还是建议使用 SerpentAI——它不仅更“轻”，部件也比较少。

14.3.4　DOTA2

DOTA2 这个游戏的出名程度不亚于《星际争霸Ⅱ》，也是一款世界知名的游戏，中文名干脆

① Virtual Network Console，虚拟网络控制台，一种类似于 Windows 远程桌面的软件。

用了音译，叫“刀塔 2”。DOTA2 的开发商是 Valve Software[①]，也就是 CS[②]的开发商——资格够老吧。

1. 环境信息

DOTA2 的 API 开放显得比较专业，只不过因为它不是开源项目，所以，习惯于开源环境的开发者在阅读其文档及下载和使用这套环境的时候，会觉得有点别扭。其地址为 https://developer.valvesoftware.com/wiki/Dota_Bot_Scripting。其中文版地址为 https://developer.valvesoftware.com/wiki/Dota_Bot_Scripting:zh-cn（如图 14-46 所示）。

图 14-46　DOTA Bot Scripting 中文版主页

一些开发 DOTA2 AI 的内行人，会推荐我们看英文版主页，因为中文版主页内容的更新速度比英文版要慢一些。

2. 业界状况

在写作本书的过程中，OpenAI 的多智能体通过自学习，在 5 对 5 团战中击败了人类玩家，如图 14-47 所示。

① Valve Software，维尔福软件公司，1996 年成立于美国华盛顿州西雅图市。
② Counter Strike，《反恐精英》。

图 14-47 DOTA2 OpenAI 机器人

2017 年，OpenAI 的机器人在 DOTA2 的 1 对 1 比赛中战胜了人类职业玩家 Dendi，在当时引起了不小的关注。记得当时一些写游戏行为树 AI 的朋友说：如果人类自己写行为树的话，由于机器人有能够连续操作、不知疲倦、出手准确的优势，其实是很容易超过人类的。然而，对于自学习的 AI 来说，还是很难的——玩过 Gym 之后相信你会有同感。在这次比赛中，OpenAI 派出的由 5 个神经网络组成的 OpenAI Five，已经能够击败 DOTA2 的业余玩家队伍了。

OpenAI Five 每天都通过自我对战来学习（形式和 AlphaGo Zero 有点像），而它每天自我对战的量是 180 年的游戏时间（因为它可以将游戏加速）。OpenAI Five 使用了由 OpenAI 提出的近端策略优化（PPO）算法的扩展版，在 256 个 GPU 和 128000 个 CPU 内核上进行训练。在训练时，每个“英雄”都使用单独的 LSTM，不使用人类的游戏数据。最终，AI 能够学会如何识别策略。

如果你没玩过 DOTA2，那就想想或者看看身边的人玩《王者荣耀》的过程吧。下面列举一下 DOTA2 的训练难点。

（1）很长的时间线

DOTA2 以每秒 30 帧的速度运行，平均时间为 45 分钟，因此，在每次运行游戏的过程中会获得约 80000 个时间片——这是一个很长的 Episode。大多数行为（例如命令“英雄”移动到某个位置）单独产生的影响较小，但个别行为（例如回城）可能会在战略上影响游戏，一些策略甚至会影响整个游戏的局面。OpenAI Five 每 4 帧观察一次，产生 20000 次移动。相比之下，一盘国际象棋通常在 40 次移动之前就会结束，一盘围棋则在 150 手之前就会结束，而且几乎每次移动（落子）都是战略性的——这也是两类游戏不一样的地方。

（2）部分观察状态

在 DOTA2 中，通过玩家的单位和建筑物，只能观察到周围的区域。地图的其他部分被隐藏了，敌人及其战略也被隐藏了。因此，玩家需要根据不完整的数据进行推断，并且需要对对手的最佳状态进行建模。相比之下，国际象棋和围棋都是信息完全显露的游戏——这又是一个很大的不同。

（3）高维连续动作空间

在 DOTA2 中，每个“英雄”可以采取数十个动作，而许多动作都是针对另一个单位或地面上的某个位置的。OpenAI 将每个“英雄”的空间分割成 170000 个可能的行动；不计算连续部分，每个时间片平均有大约 1000 次有效操作。国际象棋中的平均动作数为 35，在围棋中这个数字也只有 250（在 14.1 节讲解 AlphaGo 的时候提到过）。

（4）高维连续的观察空间

DOTA2 的观测对象（Observation）包含 10 个“英雄”、几十幢建筑物、几十个 NPC（Non-Player Character，非玩家角色）及符文、树木等，数量庞大。OpenAI 的模型通过 Valve Software 的 Bot API 观察 DOTA2 游戏的状态，观测到的状态维度数据有 20000 个左右，且其中大多是浮点数。相比之下，国际象棋棋盘上只有大约 70 个枚举值（8×8 的棋盘、6 种棋子和其他信息），围棋棋盘上则有大约 400 个枚举值（19×19 的棋盘和黑、白两种棋子）。

（5）复杂的规则

DOTA2 的游戏规则非常复杂。DOTA2 是一个已经开发了十多年的游戏，其逻辑代码有几十万行。这些逻辑代码，需要几毫秒的时间才能执行。而对国际象棋或者围棋，只需要几纳秒就能执行。DOTA2 游戏每两周更新一次，因此其环境语义会不断改变。在训练中，完全都是从自我对战中学习的，使用了 128000 个 CPU 和 256 个 P100 GPU[①]。

OpenAI 使用 Proximal Policy Optimization 的大规模版本进行学习。OpenAI Five 和 OpenAI 早期的 1 对 1 机器人都是完全从自我对战中学习的。它们从随机参数开始学习，不使用来自人类回放（Replay）的搜索或引导（如图 14-48 所示）。

RL 研究人员（包括 OpenAI 自己）一般认为，对于时间线较长的训练（就是 Episode 极长的情况），需要从根本上取得新的进展，例如分层强化学习。结果表明，如今的算法已经能够处理这些问题了——至少当它们以足够的规模和合理的探索方式运行时。在对游戏长期战略意图的把握上，OpenAI Five 已经堪比一个专业玩家。因为组建攻击塔需要时间，所以，要想获得战略地图控制等长期回报，往往需要牺牲短期回报（例如通过游戏的农业部分获得黄金）。

每个 OpenAI Five 网络都包含一个拥有 1024 个 Unit 的单层 LSTM，它可以查看当前的游戏状态（从 Valve 的 Bot API 中提取），并通过几个可能的 Action Head 发出动作。

OpenAI Five 使用观察空间和动作空间进行交互式演示。OpenAI Five 将“世界”视为 20000 个数字的列表，并通过发出一个包含 8 个枚举值的列表来采取行动。选择不同的行动和目标，就可以了解 OpenAI Five 如何对每个动作进行编码，以及它如何观察“世界”。人类看到的场景，如图 14-49 所示。

① 在比较高端的专业深度学习训练中使用的显卡，有 3584 个 CUDA 内核，16GB 显存。

	OPENAI 1V1 BOT	OPENAI FIVE
CPUs	60,000 CPU cores on Azure	128,000 preemptible CPU cores on GCP
GPUs	256 K80 GPUs on Azure	256 P100 GPUs on GCP
Experience collected	~300 years per day	~180 years per day (~900 years per day counting each hero separately)
Size of observation	~3.3 kB	~36.8 kB
Observations per second of gameplay	10	7.5
Batch size	8,388,608 observations	1,048,576 observations
Batches per minute	~20	~60

图 14-48　1 对 1 和 5 对 5 的训练复杂度对比

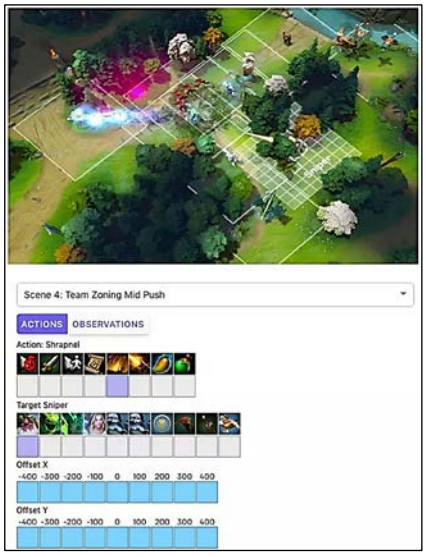


图 14-49　使用工具观察场景状况

OpenAI Five 可以对丢失的状态片段及与它所看到的相关的状态片段作出反应。OpenAI Five 的观察区域不包括弹片区域（弹片落在敌人身上的区域），人类在屏幕上能看到这些区域。OpenAI Five 能够学习如何离开（虽然不能避免进入）活跃的弹片区域，因为当进入弹片区域时，它可以看到自己的“健康”状况正在下降。

最有趣的是，在一个 5 对 5 的环境中，OpenAI Five 还能学会 GANK[①]，如图 14-50 所示[②]。

图 14-50　机器人团战 GANK

惊不惊喜，意不意外？这是很多号称玩了两三年的游戏老手都不见得能养成的意识，而机器人自学成才了。

即便是人类这种真正的智能体，也不一定有足够的精力和意识来实施一场成功的 GANK，可 OpenAI Five 学会了。OpenAI Five 学会 GANK 的意义，在我看来，和 AlphaGo Zero 能够学会围棋定式的意义相似，既是一种与人类起源不同的生命或者智慧进化到一定程度后的必然，也是一种无师自通地达成高级任务中所需要的战术性局部套路的总结能力。在一定程度上，这是机器人智慧的表现。以前我们说机器人不会思考，是因为环境单纯，机器人使用的技术和方法偏重于人类在编写程序时设置的技巧（使用这些技巧可以加快树遍历的进程、提高估值的准确性）。这种现象在小规模、近距离观察 Agent 的过程中体现得非常明显，并且显得非常“不智能”。

环境和规模一旦扩大，利用硬件优势让机器人自己进行“推理”的优势就显现出来了。虽然在计算层面的处理中，树搜索、估值回溯、梯度上升等技巧仍在使用，但在宏观上体现出来的是一个能够通过不断总结进行推理的过程——尽管速度有些慢，但仍然给了我们这些爱好者很多的鼓励和信心。相信随着时间的推移，会有更多优秀的算法模型被研究出来。结合摩尔定律每 18 个月计算能力提升 1 倍的“魔咒”，最终能不能研发出“终结者”我不知道，但我敢肯定，未来的世界一定非常精彩！

14.4　本章小结

本章的内容看上去有点杂，但是杂而不乱。

对著名的 AlphaGo 和 AlphaGo Zero 的整体思路，我们进行了详细的讲解。不过，了解一下就可以了，要想真的训练出一个 AlphaGo Zero，恐怕要使用超过 170 个 TPU[③]——一个 TPU 的计算

① GANK 是 MOBA 类游戏中的一种常用战术，指通过一个或几个游戏角色的行动，对对方的游戏角色进行偷袭、包抄、击杀，或者以人数或技能方面的优势，有预谋地击杀对手，从而起到压制作用。

② 引自新智元微信公众号：https://mp.weixin.qq.com/s/rPsxvzOLuoo_IRtgH94Ecw。

③ 引自新浪科技：http://www.guancha.cn/kegongliliang/2017_05_25_410107.shtml。

能力相当于一个 GPU 计算能力的 15 ~ 30 倍[①]。训练这样一个网络所需要的资金，个人和普通公司恐怕难以拥有。可是，毕竟我们已经处在深度强化学习一展风采的时代，如果不了解 AlphaGo 和 AlphaGo Zero 的理念，就好像到北京旅游没去长城一样——莫名遗憾。

随后介绍的几个高级“玩具”，看上去“友好”了一些，而且具象性比较强。至少在我周围有很多朋友认为，围棋是最难的游戏项目。我想，这可能只是他们的直观感觉吧，估计在他们身边玩《王者荣耀》的人比下围棋的人多。客观地说，实现 DOTA2 的智能机器人的难度比围棋高得多，甚至可以说，高了不止一个两个数量级。所以，DeepMind 挑战人类玩家所用的 AI，在 1 对 1 Solo[②]上的速度还算比较快，在 5 对 5 时却等了很久——还是用 CPU 和 GPU“堆”出来的。

“玩具”就介绍到这里，大家视自己的财力和时间去选择游戏对象吧。如果非要说技巧，我觉得，需要注意一点：我们“教”给计算机的那些认知原则和人类的认知原则其实是一样的，那就是化繁为简。如果输入的维度多，在一定程度上就可以认为，输入维度中的大部分其实没有携带那么多信息——信息是非常稀疏的。对这类微观世界的认知，可以参考人类对微观世界认知的技巧。例如，如果人类对客观世界的建模一上来就处在分子级别，那么一切描述的对象都是分子级别的，而数量巨大的分子会让观测显得非常低效。事实上，人类为了让认知过程更为有效，会尝试用宏观的概念来描述微观的现象。例如，我们不仅可以用分子的平均动能和势能来描述水的内能，也可以绕开这些概念，改为用温度来描述，这样，不管是在观测手段层面，还是在概念层面，都显得“清爽”了很多。再如，对一个地区的人的收入水平，可以用每个人每月的收入和支出来描述（这相当于原始的像素级别的输入），如果用宏观的统计值来描述，即一个地区的平均月收入及最大值、最小值、方差、样本数，就可以将输入维度减少到 5 ~ 6 个，这样做尽管丢掉了一些细节，但突出了特征。在经济学研究中，学者们也喜欢用各种指标对数据进行“浓缩”，借以指代某方面的特征或者某些方面的综合抽象特征。降维绝对是一个可以利用的好方法，我们可以充分借鉴。

① 引自观察者网：http://www.guancha.cn/kegongliliang/2017_05_25_410107.shtml。

② 原意为独唱、独奏，在竞技类游戏中特指单挑。

第 15 章　扩展讨论

到第 14 章为止，我们仔细讨论了几乎所有的强化学习算法和深度强化学习算法。能给出的代码，都放在本书的 GitHub 页面中了，读者可以直接下载。此外，我们讨论了众多相关领域的顶尖案例，认识了人工智能爱好者们为我们准备的各种高级“玩具”。

在本书的最后一章，我们放松一下，聊一些看上去“高大上”但仅用于扩展思维的话题。

15.1　TRPO

“TRPO”的全称是“Trust Region Policy Optimization”，在 2017 年由 John Schulman 等人以同名论文的形式发表。“TRPO”译为“置信区域策略优化”，顾名思义，就是在一个“置信”的“区域”内做策略优化。

区域有什么置信或者不置信的呢？什么是不置信的区域呢？对这些问题，要从策略梯度优化的公式说起。

$$\theta_{\text{new}} = \theta_{\text{old}} + \alpha \nabla_{\theta} J.$$

θ_{new} 是更新后的新网络的参数，θ_{old} 是更新前的旧网络的参数，$\nabla_{\theta} J$ 是评价函数的梯度，α 是一个类似于学习率的东西——步长。这个步长设置为多少合适呢？设置得太小，学不到东西；设置得太大，容易一步迈到“坑”里，产生模型迭代进化中的颠簸，甚至使模型变得越来越差。

更新的量应该是多大，是有计算技巧的，计算新老策略之间的 KL 散度就是方法之一。形象地理解，就是计算两个策略之间的“距离”。只要它们之间的“距离”在一个合理的范围内，就可以保证在一个安全的范围内进行策略更新。这个“距离”只用于保证一件事，那就是更新后的网络所产生的评价函数值不能比更新前的小。或者，要先计算出更新多少后策略仍在“进化”而非“退化”，再做更新。

在 *Trust Region Policy Optimization* 这篇论文中对 TRPO 进行了严格的推导和论证。如图 15-1 所示，这幅图形象地说明了，使用 TRPO 方法对 Gym 里的游戏进行优化所产生的效果是非常好的，在更新过程中，大都可以获得单调上升的奖励值。“Single Path”和“Vine”是 TRPO 算法的两个亚种。“CEM”是“Cross-Entropy Method”的缩写。“CMA”是“Covariance Matrix Adaption”的缩写。“Natural Gradient”是一种经典的策略梯度算法。在 Cartpole（小车立杆子）、Swimmer、Hopper、Walker 这几个游戏中，TRPO 的两个算法亚种有比较明显的优势。

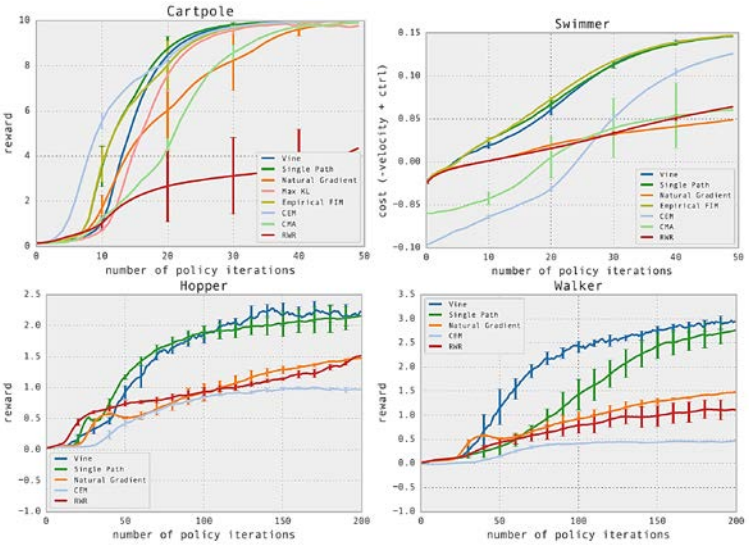


图 15-1　TRPO 产生的回报值

不管是用 TensorFlow 还是用 PyTorch，都可以做 TRPO 的实现。准确地说，用什么框架不重要，重要的是计算 KL 散度和更新步长。

对 TRPO，我自己没有做过实现，GitHub 上做相关实现的人也不多，所以给大家一段代码作为参考：https://github.com/ikostrikov/pytorch-trpo。这里不仅有 TRPO，还有 PPO（Proximal Policy Optimization，近端策略优化），有能力或者感兴趣的读者朋友可以试试看。

15.2　反向强化学习

反向强化学习是近年来比较热门的研究课题。其英文名为“Inverse Reinforcement Learning”，也被翻译成“逆向强化学习”。在强化学习中，$R(s,a)$ 是较难确定的超参数之一，我们称之为奖励值。反向强化学习研究的是如何通过一定的方法把这个值“学”出来。

奖励值是一个重要的超参数，因为这个值设置得不好，会直接导致状态值的估算不准确，从而把模型误导到其他目标上去了。说“误导”都是好听的，实际上会“坑死”机器人。所以，需要把这个值尽可能估准——最终目的是让模型顺利进化。

有这样一种说法：专家（主要指人类或者示范类）在完成某项任务时，其决策往往是最优的或者接近最优的。在做游戏演示的时候，其实展示的是一次接近“神操作”的“Show”，它可能是游戏达人在玩游戏的时候“录下来”的，也可能是游戏的创作者为了给玩家提供一些具有观赏性的内容而特别编写的一段“优美”的脚本——不管怎样，这些策略都是非常优秀的。这类操作作为样本，是非常优秀的先验知识。那么，我们是不是可以假设：当所有的策略所产生的累积回报期望都不如这些专家策略所产生的累积回报期望大时，强化学习所对应的回报函数就是根据刚才

所说的这些示例学到的回报函数。因此，反向强化学习可以定义为从专家示例中学到回报函数。当然，本书涉及的实验环境，大都不会用到如此高深的方法，一般通过比较简单的方式（例如得分）就能很快确定 $R(s,a)$ 的大小了。

反向强化学习方法有哪些呢？有学徒学习（Apprenticeship Learning）、MMP 方法（Maximum Margin Planning）、神经逆向强化学习（Neural Inverse Reinforcement Learning，NIRL）、结构化分类（Structured Classification for Inverse Reinforcement Learning，SCIRL）等，每种方法都有若干子分类。在入门级学习中，了解一下学徒学习就可以了。

学徒学习是由 Pieter Abbeel[①]和 Andrew Ng[②]在 2004 年提出的。其大致思路是，假设一个待定的估值函数

$$R(s) = \boldsymbol{w}\phi(s).$$

$\boldsymbol{w}$ 是待定系数（应该是一个矩阵）。$\phi(s)$ 是一个基函数（你认为它是一个神经网络也没关系，因为神经网络也能实现相同的计算逻辑）。然后，给一个 m 帧的专家操作，得到这个操作的估值。同时，给一个待优化的策略，在这个策略下也会有相应的状态估值。设置一个误差阈值 ε，比较两个估值，如果两者的差比 ε 小，就可以终止了。这时，$R(s) = \boldsymbol{w}\phi(s)$ 就是指奖励值的大小。如果误差大于等于 ε，就要接着优化。

从形式上看，感觉和监督学习有点像。是的，有一种很像监督学习的方式，也采取了“看录像，学手段”的形式，叫作模仿学习（Imitation Learning）。

模仿学习是为了避免机器人在冷启动状态下在极为广阔的空间中开着“脑洞”去做各种尝试而设计的，应该说，是对机器人训练手段的一种有益补充。研究机器人的目的是让技术落地，而不是让机器人“无师自通”地去学习——万一“通”不了，可怎么办呢？总之，要让机器人以 Baseline（基线）为基础开始进化。反向强化学习中的一些基础手段就能帮我们处理这方面的需求。

15.3　模型压缩

模型压缩一直是深度学习中的一个独立的领域，也是一个非常有趣的研究方向。为什么要压缩深度学习模型呢？主要因为深度学习的模型通常层数很多、参数数量很大、执行速度很慢，而这样的模型产生的巨大能耗对嵌入式设备那疲弱的负载能力来说，简直是“不能承受之重”。我们都很清楚，汽车自动驾驶机器人、工业人形机器人等尖端产品，除了要让机器人能够以高精度完成指定任务，还要有足够的实时性。例如，汽车自动驾驶机器人对外部摄像机的处理能力需要达到 30FPS，也就是要接近甚至超过人的反应速度。

这意味着什么呢？为了提高神经网络的精度，我们使网络变得更深，为网络增加了参数，这

① Pieter Abbeel，伯克利大学教授，曾在 OpenAI 任职，COVARIANT.AI 的创始人。
② 吴恩达教授，著名美籍华裔计算机科学家，斯坦福大学教授。

样做确实能在一定程度上提高网络的精度。可是，“是药三分毒”，在获得网络精度提升的同时，无疑会带来一些负面影响。例如：大量的参数意味着大量的运算，这会极大地消耗 CPU 资源；大量的参数也意味着大量的空间占用，这会极大地占用内存资源。CPU 和内存在嵌入式平台上绝对属于稀缺资源。因为嵌入式平台通常是由锂电池或者一些低功率的电源驱动的，所以 CPU 几乎无法在“酒足饭饱”的情况下工作。内存也是嵌入式平台上的稀缺资源，8GB 内存对移动设备来说已经非常奢侈了。

看一组参考数据：用 Intel i7 2.5GHz 的 CPU 来做 TensorFlow SSD[①]这样的模型，在 320 像素 × 240 像素的图片上为 21FPS，在 640 像素 × 480 像素的图片上仅为 11FPS——这不是在嵌入式设备上工作的数据，而是在普通 PC 上工作的数据（使用嵌入式设备的情况会更糟）。在汽车自动驾驶机器人领域，还没有普遍使用强化学习（或者说，没有完全使用强化学习）。但是，对于目标识别任务，SSD 这种复杂的技术还是非常有可能作为一个组件来使用的。不论 SSD 的后端做目标提取时使用的是 VGG 还是 ResNet，计算量都非常大。相比之下，用在 Atari 游戏上的 DQN、DDPG 这类网络就显得太小了。有朝一日，如果你需要将一个 SSD 和一个 DQN 一起移植到一个嵌入式环境中，想必也应该优先用模型压缩的方式来优化 SSD。

深度学习模型体积庞大、计算速度慢，主要是模型中的参数数量太多所致。以如图 15-2 所示的 AlexNet 为例，这个模型中有 60M[②]个参数，按照每个参数占用 4Byte 计算，足有 240MB——已经超过了很多移动端 App 的大小。将如此“庞大”的深度学习模型嵌入一个 App 并完成相关的工作，其结果是灾难性的：一来会让很多人望而却步，不愿意下载这种嵌入了深度学习模型的 App；二来会在运行过程中极大地消耗计算资源。

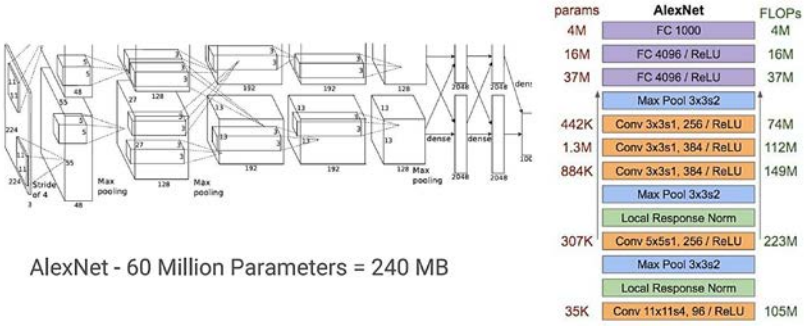


图 15-2　AlexNet 的参数文件大小

关于计算资源，我们通常用一个计量单位表示，就是图 15-2 右侧的“FLOPs”。

FLOPs 是一种用于标记计算规模的单位。将浮点数先乘后加，算出一个 FLOP。如果一个基

① Single Shot Multibox Detector，数据来源：https://towardsdatascience.com/how-to-build-a-real-time-hand-detector-using-neural-networks-ssd-on-tensorflow-d6bac0e4b2ce。

② $1M=10^6$。

本浮点数乘法运算是一个 FLOP，那么 1×1 是 1 FLOP，$2 \times 3 + 1 \times 2 + 3 \times 3 + 5 \times 10$ 是 4 FLOPs。假设有一个卷积层，其参数为 $n \times (h \times w \times c + 1)$，$h$ 是卷积核的长度，w 是卷积核的宽度，c 是输入通道数，n 是输出通道数，输入的 Feature Map 尺寸为 $H' \times W'$，那么

$$\text{FLOPs} = H' \times W' \times n \times (h \times w \times c + 1).$$

105M 的 FLOPs 就是 1.05 亿个 FLOP——这个数字相当大。

关于模型压缩的论文非常多，而且每年都有新的论文发表。由于本书篇幅有限，下面我们就从几个压缩思路出发，给大家一些提示。其实，模型压缩还是沿着稀疏矩阵、量化、结构优化这样的思路来实现的。

15.3.1　剪枝

“剪枝”的英文名叫“Pruning”。Pruning 的思路大致是这样的：一个神经网络，尤其是全连接网络，有非常多的权重，这些权重实际上造成了大量的冗余，如图 15-3 所示。也就是说，即使不使用这么多的神经节点和权重，也能把这个网络本身的分类能力或者回归能力实现个差不离。参照第 12 章 NEAT 训练网络的过程，就能大致理解其思路了。那么，我们怎样才能找到那些冗余的权重呢？

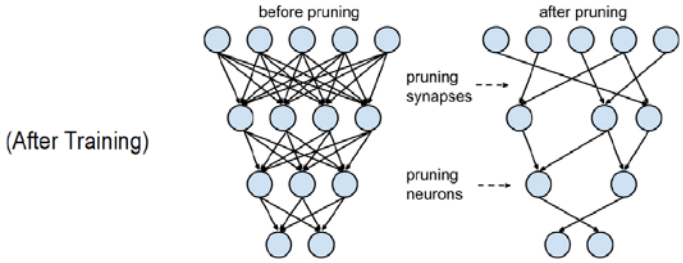
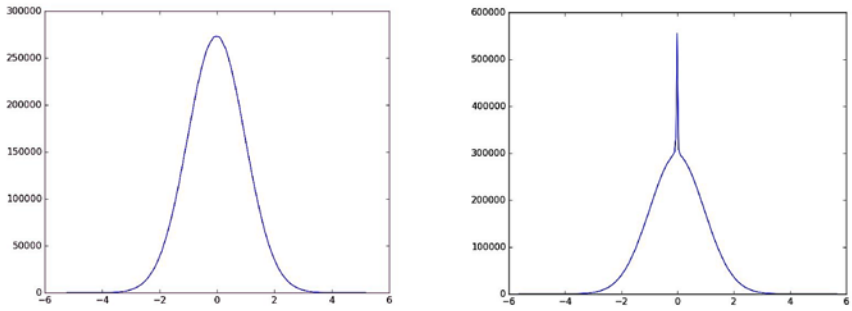


图 15-3　深度学习网络 Pruning 示意图

如果矩阵可以被稀疏化，那么最简单的方法就是去掉那些值为 0 的权重。这种思路有点像无损压缩里面的压缩算法。例如，在一段文字中找到大量重复的内容，用尽可能短的编码来代替这些重复的文字，再做一个数据字典，在解压缩时通过查表的方法进行还原即可。

同样，如果能够在一个网络中找到那些小于某些阈值（Threshold）的权重并把它们去掉，就能节省部分存储空间。确实不需要存储那么多极为接近 0 的权重，因为 0 乘以任何数的结果都是 0——宝贵的存储空间用来保存这些 0 实在有点浪费。因此，可以考虑用一些优化方法来保存这些稀疏的矩阵（几乎所有的压缩算法都可以应用在这个场景中）。

例如，加入正则化惩罚项 L1 和 L2，如图 15-4 所示。我可以明确地告诉大家，L1 惩罚项的效果通常更好。



L2 惩罚项　　L1 惩罚项

图 15-4　权值分布

在第 6 章中，我们专门介绍了 L1 和 L2 两个不同的正则化惩罚项对权重的影响，当时的目的主要是防止过拟合。现在，第二个目的出现了：加入 L2 惩罚项，可以使权值变小；加入 L1 惩罚项，会在极大程度上使很多值极为接近 0。这是由惩罚项本身的特征造成的，损失函数的损失等高线与惩罚项曲线相切，L1 惩罚项的切点可以让权值“贴”在坐标轴上，但 L2 惩罚项不行。这就是 L1 惩罚项能够产生更多极为接近 0 的权值的原因。

所以，如果使用了带有 L1 惩罚项的损失函数，在训练出来的全连接网络层（或者卷积神经网络层）中，都会展现如图 15-4 右图所示的特性——太好了，绝对值小于某个很小的值的那些权重都可以被“无视”了，反正乘或不乘对计算结果的影响微乎其微，这样空间就节省出来了。不过，这里有两个问题需要大家注意。

- 矩阵会变得稀疏，因此会节省磁盘空间。但是，如果矩阵保存在内存中，那么仍然会占用相同的空间。这是因为，即便对一个数字 0，如果用 Float32 类型进行保存，仍然会占用 4Byte，而这意味着，在将这个模型从磁盘还原到内存，准备进行计算的时候，仍然需要占用与压缩前相同的空间。
- 矩阵计算的速度不会加快，因为每个 0 权重仍然要占用空间并参与运算。不管是用 PyTorch 框架、TensorFlow 框架，还是自己手动编写正向传播代码，都很难避开这个问题。所以，即使是一个简单的正向传播操作，对于优化的稀疏矩阵乘积算法，也需要进行稀疏化编程处理（难度还是蛮大的）。

15.3.2　量化

“量化”的英文名为“Quantization”。这个词的本意是，通过一些计算手段把原先在连续值域上分布的数值用离散值域中的数值来表示的技术过程。其实，这仍然是一种稀疏化的手段。简单地说，就是原来网络中的成千上万个参数都是连续的数值，现在我们要通过一些手段让它们都取一个离自己最近的量化值。

例如，有 0.001、0.002、0.003、0.002、0.010、0.009、0.011、0.009 这样几个值。可以看到这

里有 8 个不同的浮点数，但是基本上聚成了两堆，一堆在 0.002 附近，另一堆在 0.010 附近。现在我们有个大胆的想法：能不能用 0.002 代表第一堆的值，用 0.010 代表第二堆的值，这样原本需要 8 个 4Byte 的值就可以只用 2 个 4Byte 的值来表示了（用一个长度为 2 的数组保存，一个下标是 0，另一个下标是 1）。然后，原来的数组就可以表示为 [0]、[0]、[0]、[0]、[1]、[1]、[1]、[1]。由于只有两个值，保存这个新数组只需要 8bit，也就是 1Byte。这样，8Byte 的值只需要 3Byte 就能保存了，这就是一种压缩的思路。

这里会有一个量化的误差，就是原来的 0.001、0.002、0.003、0.002 用 0.002 来表示了，那么有两个值有 0.001 大小的误差，而这个误差是不得不考虑的。所以，还需要对每个权重做一个偏移量的量化——对权重与量化值之间的差，也尝试用这种量化方式逐步量化。只要方法得当，大都能在压缩率和精度之间找到一个平衡点（毕竟这属于典型的有损压缩，对这种差距，我们必须心里有数）。

15.3.3　结构压缩

与前面两种方法不同，结构压缩的思路大部分不是针对压缩后的模型文件的，而是针对网络结构本身的。对主流的深度神经网络结构来说，能否考虑使用更为简单或者紧凑的结构来代替原有的复杂结构呢？这就是结构压缩思路的本源。可以在训练好的模型上进行压缩处理，也可以设计一些小的 CNN 结构单元来进行压缩处理。

GoogleNet 其实就是一个非常好的例子。可以说，GoogleNet 是第一个从优化和利用网络内部计算资源的角度来进行改进的网络，也就是我们常说的 NIN（Network in Network）——一个大网络由很多的小网络结构组成。GoogleNet 比起它的前一代深度学习网络 AlexNet，有很多的改进。在 GoogleNet 中，有参数的网络层达 22 层，这使其在分类任务的精确度上有显著的提升，而其参数的数量仅为 AlexNet 的 $\frac{1}{12}$。更厉害的是，GoogleNet 的计算代价比 AlexNet 少一半——名副其实的事半功倍。

GoogleNet 利用的结构，我们习惯称之为 Inception，如图 15-5 所示。

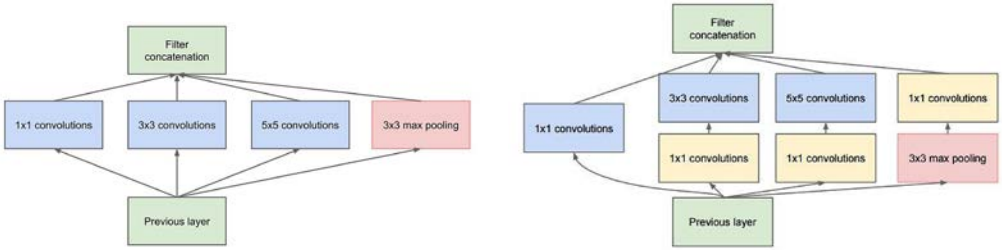


图 15-5　Inception 结构

这两个都是 Inception：左边的是旧版本的 Inception，学名为“Naïve Approach Inception”；右

边的是新版本的 Inception，学名为“Dimensionality Reduction Inception”。

找到这两个结构的区别了吗？细心的读者朋友可能会发现，新版本的 Inception 虽然和旧版本的 Inception 一样，有多个卷积“旁路”，可以通过不同尺寸的卷积核，但新版本的 Inception 显然要“厚”一些，有两层，并且加入的卷积层是 1×1 的卷积核。这不是无端地往里面掺东西，而是有原因的。我们来计算一下。

一个卷积层的参数数量应该是多少呢？如图 15-6 所示，应该是

$$C_{\text{inp}} \times W \times H \times C_{\text{out}}.$$

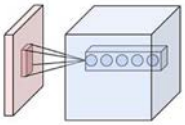


图 15-6　卷积核示意图

例如，输入为 1 个通道，输出为 5 个通道，卷积核的长和宽都是 3，那么需要多少个参数呢？$1 \times 3 \times 3 \times 5 = 45$，一共需要 45 个参数。每扫描一次需要进行多少次 FLOPs 计算呢？只要有一次 Stride，就要做 9 次乘法和 9 次加法才能产生一个卷积结果——这还没考虑后面的激励函数的计算需求。而如果采用小的卷积核，例如 1×1 的，这个值立刻就降下来了，$1 \times 1 \times 1 \times 5 = 5$，仅有 5 个参数值，一次 Stride 仅需 1 次乘法和 1 次加法就可以完成卷积任务了。所以，从这个角度看，越小的卷积核对计算资源的消耗越少。当然，在这里没有讨论卷积核变小对提取能力的影响，但从实验结果上看，影响不明显。对提取能力的影响不明显，却明显影响（提高）了计算效率，这当然是一种技术进步。

那么，这两种 Inception 的参数数量差距为多少呢？

Naïve Approach Inception：

$$c_{\text{inp}} \times \big((1 \times 1) + (3 \times 3) + (5 \times 5)\big) \times c_{\text{out}} = 35 \times c_{\text{inp}} \times c_{\text{out}}.$$

当 $c_{\text{inp}} = 512$，$c_{\text{out}} = 1024$ 时，上式应该等于 18350080。

Dimensionality Reduction Inception：

$$2 \times c_{\text{inp}} \times (1 \times 1) \times \text{cout}_{\text{inp}_{\text{out}}} \text{int}_{\text{int}}.$$

当 $c_{\text{inp}} = 512$，c_{int}，$c_{\text{out}} = 1024$ 时，上式应该等于 2818048。

前者是后者的 6.5 倍左右，压缩率还是很高的。也就是说，在 NIN 这种结构下，有进一步压缩的空间。

使用 NIN 的思路对网络进行优化的例子还有不少，比较著名的是一个叫作“SqueezNet”的网络结构，尽管其大小只有 4.8MB，但可以实现和 240MB 的 AlextNet 几乎相同水平的分类能力。压缩后的 SqueezeNet 仅有 0.52MB，也就是 AlexNet 的 $\frac{1}{460}$ 左右，它具有和 AlexNet 相同的分类

能力。这足以说明，NIN 是一种成熟、有效且可以显著减小深度学习模型尺寸的研究套路，值得我们持续关注。

在这种思路的引导下，有很多网络涌现出来，它们中的大部分也是通过优化卷积层结构实现的。例如，2017 年谷歌提出了一种叫作“MobileNet”的网络结构，表现不俗。MobileNet 像深度残差网络一样，提出了一种对卷积神经网络结构的改进，而对任务和用途没有做特殊限制（如图 15-7 所示）。既然深度残差网络可以用来做分类，可以用来做目标检测，可以用来让 AlphaGo 学习动作函数，那么 MobileNet 也可以用来做这些工作——反正目的是通过多个卷积层提取特征并做非线性滤波，这些本质性的操作是没有任何改变的。

Table 8. MobileNet Comparison to Popular Models

Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6%	569	4.2
GoogleNet	69.8%	1550	6.8
VGG 16	71.5%	15300	138

图 15-7　MobileNet 与传统卷积神经网络的对比

MobileNet 与传统网络相比，例如与 GoogleNet 和 VGG-16 的分类任务相比，精确度几乎是相同的（差距非常小），甚至比 GoogleNet 要好一些。在参数数量方面则相差悬殊，VGG-16 有 1.38 亿个参数，MobileNet-224 只有 400 多万个参数，相差约 32 倍（也就是说，文件尺寸相差约 32 倍）。在计算量方面，VGG-16 为 153 亿次，MobileNet-224 为 569 万次，相差约 27 倍（估计耗电量也要差这么多）。这对于在嵌入式设备上运行的神经网络来说，无疑是非常有意义的——这是能用与不能用的区别。照这个趋势发展下去，应该说，我们还是可以期许卷积神经网络在移动设备上的普及应用的（也许只要几年时间就可以实现了）。当然，这仍然取决于硬件技术的发展，深度学习工程人才的储备与培养，以及读者朋友在各行各业的不断努力与探索。

还有一些对数学技巧要求比较高的方法，例如 SVD（Singular Value Decomposition，奇异值分解），如果自己实施，可能会面临很多数学和工程学方面的问题。在这里，我不建议大家强行挑战这些项目，有能力或者感兴趣的读者可以通过相关资料了解一下。对关注应用层面的读者朋友，我的建议是持续关注行业动态，尤其是新的微型化模型方面的行业动态，因为总有很多聪明的计算机科学家致力于不断研究模型压缩的通用技术框架和个案性的模型。

这个世界一定会越来越美好。

15.4　本章小结

这一章是本书正文部分的最后一章了。在这一章中，我们聊了一些有趣的扩展话题，包括 TRPO、反向强化学习及模型压缩。对训练有帮助的是 TRPO 和反向强化学习，对落地有帮助的是模型压缩。

对于模型压缩，即便不使用强化学习方法训练一个机器人，只要最终使用的是一个工作在嵌入式环境中的机器人，就十有八九会用到相关的知识。希望这些内容能对大家的日常工作有所帮助。应该说，在对前沿趋势的研究中，“一大”“一小”两个方向仍在发展。

“一大”就是指模型变大、变深、变复杂，处理样本量更为巨大的环境。对应地，会催生更大规模的集成电路设备，例如 TPU 和高效能的 GPU。这些都会帮助我们在越来越多的应用技术领域游刃有余地进行深度学习研究——谷歌的股票可以长期持有，错不了。

“一小”就是指模型的小型化、结构化的调整和优化。以牺牲一定的精确度来换取数十倍的速度优势，这种技术思路应该会长期处于主导地位。模型压缩在移动设备上的应用，在未来会持续发展，只不过目前相关人才太少。相关人才一旦多起来，就一定会出现井喷式的爆发。这一天值得期待！

后记

书写到这里，就要跟读者朋友们说再见了。作为一个强化学习的爱好者、研究者，一个“伪强化学习专家”，我在这几年里也尝试用深度强化学习做过不少实验，例如尝试用它玩游戏、炒股票，尝试用它做广告投放……在一些场景中确实得到了令人欣喜的结果，但也有很多情况是在消耗了大量的训练时间后只取得了一点微不足道的进步。但是，不管是成功带来的喜悦，还是挫折带来的沮丧，都是托举我和我的团队乘风前进的浪花。

深度强化学习确实是一个很有难度的领域——确实。例如，样本的处理技巧，Transition 的处理技巧，深度学习的学习率，更新方式的处理技巧，多任务、多限制条件的处理技巧……在很多地方，都处在以试探为主的研究范畴，都有大量 Trick 可以尝试和对比。

不友好的地方是，深度强化学习由于没有理论层面的重大突破，仍旧是一个对 CPU 和 GPU 的消耗非常大的“杀手级”研究领域。所以，在这个领域，如果沿着当前的研究方向走，需要大量的计算资源，这意味着需要大量的资金。要想迈过这个门槛，需要一定的经济基础。

发展还在继续，通往未来的路虽然崎岖、漫长，但仍在延伸。我和我的团队也非常欢迎读者朋友参与研究和探讨，为了使以强化学习为手段的人工智能技术在更多的领域内发挥更大的价值与作用而共同努力。

最后，祝每一位读者朋友学习顺利！

附录 A

A.1 安装 Ubuntu

本书的实验环境选择了 Ubuntu 17.10，代码也在该版本的 Linux 平台上进行了测试，因此推荐大家选用该版本。

选择安装“Install Ubuntu”，如图 A-1 所示。

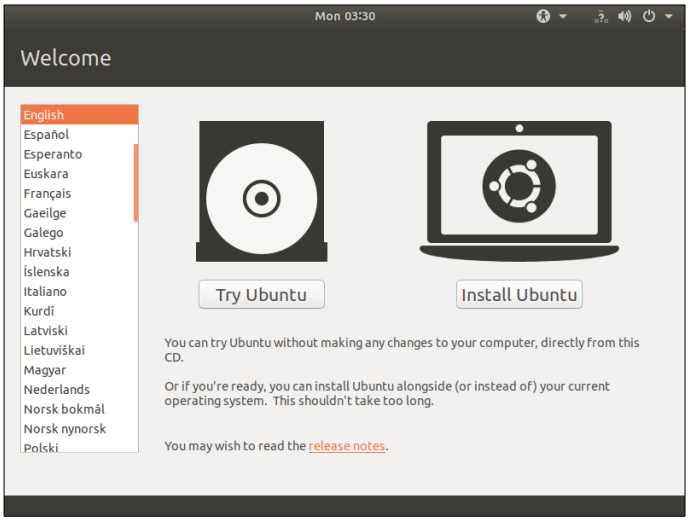


图 A-1 安装 Ubuntu

一切后续需要的软件组件，我们都可以通过 Shell 命令来安装，所以在现阶段可以不选择任何选项，直接单击“Continue”按钮，如图 A-2 所示。

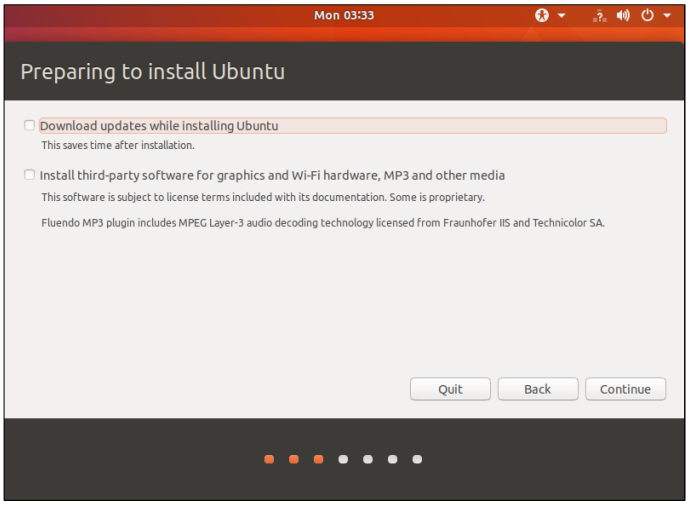


图 A-2　是否下载或更新软件

在 PC 裸机或可覆盖磁盘上全新安装的情况下，单击选中“Erase disk and install Ubuntu”单选按钮，然后单击“Install Now”按钮，如图 A-3 所示。

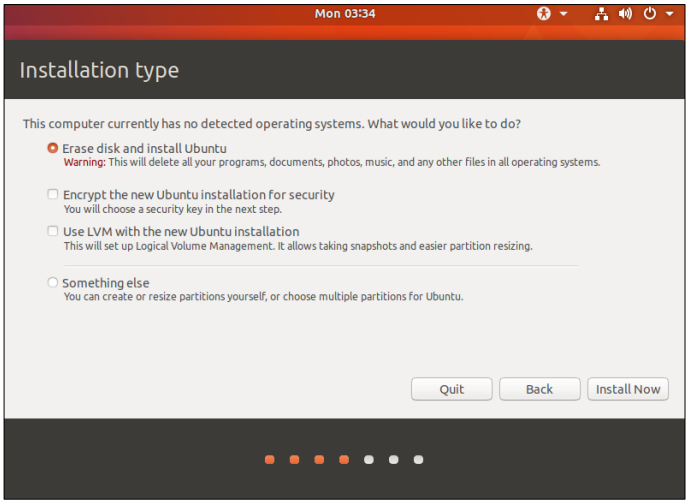


图 A-3　选择安装类型

确认安装类型，单击“Continue”按钮，如图 A-4 所示。

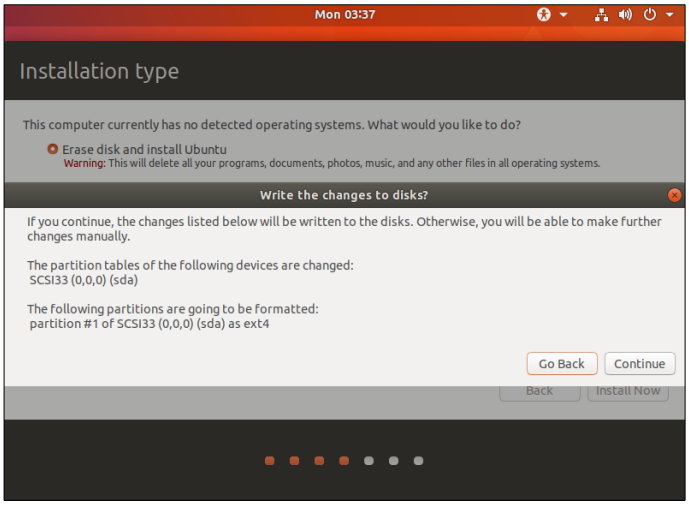


图 A-4　确认安装类型

选择“Shanghai”（上海）所在的东 8 时区，然后单击“Continue”按钮。

选择“English (US)”（美式英文键盘）选项，然后单击“Continue”按钮，如图 A-5 和图 A-6 所示。

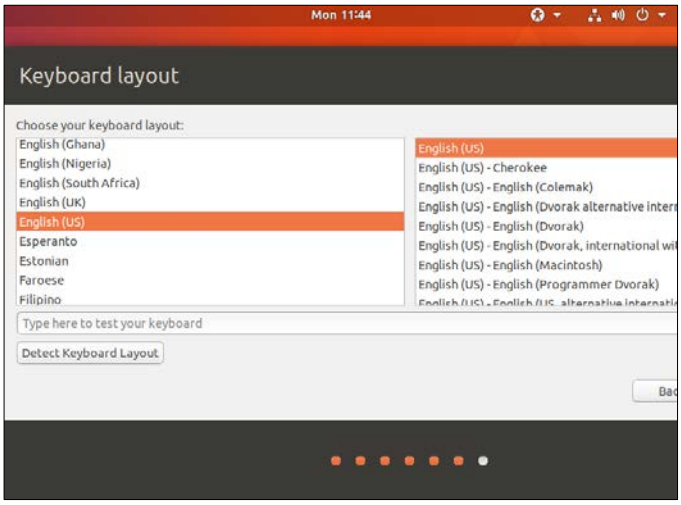


图 A-5　选择键盘类型（1）

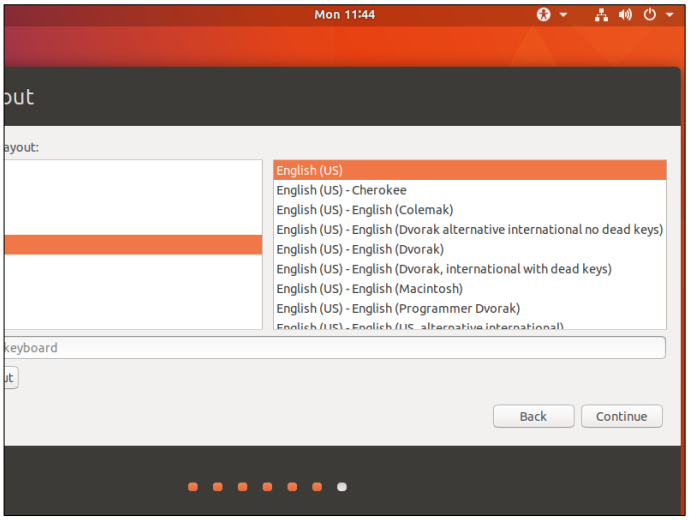


图 A-6　选择键盘类型（2）

创建用户，根据个人情况输入相应的英文用户名。此处我输入的用户名为“gao”，主机名为“gao-host”，如图 A-7 所示。不过，由于在这里基本不涉及服务器多机通信与开发的问题，所以这个用户名不会影响实验结果。密码可以任意输入。由于是在个人环境中做实验，我将密码设置为“123456”，可以看到旁边的提示信息为“Fair password”（普通密码）。这种密码强度在服务器环境中不够安全，在单机实验中可以忽略这个问题。然后，单击“Continue”按钮。

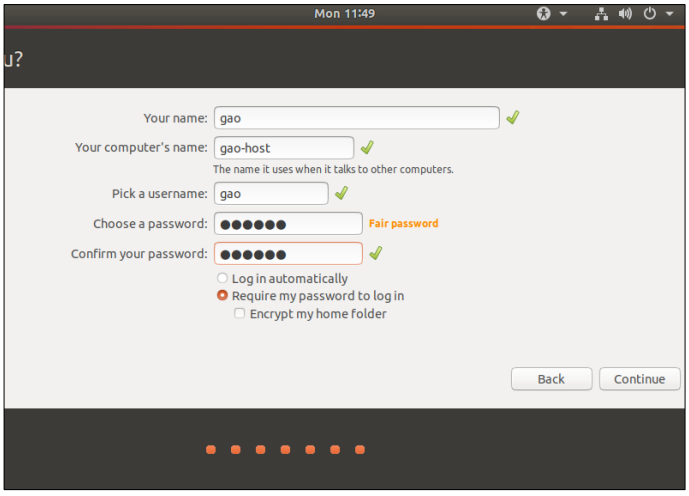


图 A-7　创建用户

安装过程不会太长，性能比较好的计算机几分钟就可以完成，如图 A-8 所示。耐心等待就可以了。

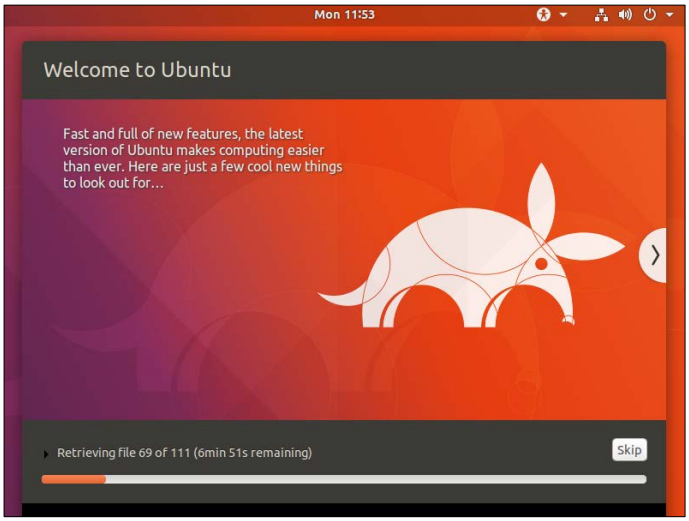


图 A-8　Ubuntu 安装过程

单击“Restart Now”按钮，重启计算机，如图 A-9 所示。

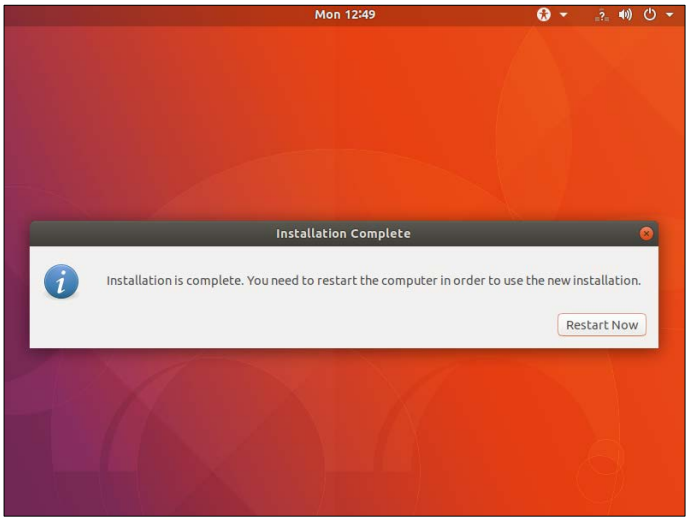


图 A-9　重启计算机

如果你更喜欢使用 CentOS 系列的 Linux 操作系统，也是可以的（桌面版比较方便做实验）。

A.2 安装 CUDA 环境

“CUDA”的全称是“Compute Unified Device Architecture”，它是显卡厂商 NVIDIA 推出的运算平台。一块 NVIDIA GPU 显卡在安装相应的 CUDA 驱动后，就可以通过 CUDA 强大的功能，并行使用 GPU 中的 CUDA 内核来训练神经网络了。

不同的 GPU 芯片型号和版本，需要与其对应的 CUDA 版本才能正确驱动，装错版本将导致无法正确驱动（具体的对应关系请访问 NVIDIA 官方网站查询）。在这里，以安装 CUDA 8.0 和 CUDNN 5.1 为例讲解。

首先，下载 CUDA 8.0 deb 文件，具体如下。

```
wget https://developer.nvidia.com/compute/cuda/8.0/Prod2/local_installers/cuda-repo
-ubuntu1604-8-0-local-ga2_8.0.61-1_amd64-deb
```

```
wget https://developer.nvidia.com/compute/cuda/8.0/Prod2/patches/2/cuda-repo-ubuntu
1604-8-0-local-cublas-performance-update_8.0.61-1_amd64-deb
```

接着，安装 CUDA 8.0 deb 文件，具体如下。

```
sudo dpkg -i cuda-repo-ubuntu1604-8-0-local-ga2_8.0.61-1_amd64.deb
```

更新安装源文件，安装 CUDA，具体如下。

```
sudo apt-get update

sudo apt-get install cuda
```

下载 CUDNN 5.1 的压缩文件，具体如下。

```
wget https://developer.nvidia.com/compute/machine-learning/cudnn/secure/v5.1/prod_2
0161129/8.0/cudnn-8.0-linux-x64-v5.1-tgz
```

解压缩，具体如下。

```
tar zxvf cudnn-8.0-linux-x64-v5.1-tgz

cd cuda/include
```

复制头文件，并给头文件添加链接，具体如下。

```
sudo cp cudnn.h /usr/local/cuda/include

cd ../lib64

sudo rm -rf libcudnn.so libcudnn.so.5

sudo ln -s libcudnn.so.5.1.5 libcudnn.so.5
```

```
sudo ln -s libcudnn.so.5 libcudnn.so
```

当你正常启动 Python，并且可以正常导入 PyTorch 的 GPU 版本软件包时，就说明 CUDA 已经安装好了。

请注意，CUDA 环境并不是显卡驱动。如果要在 Ubuntu 环境中安装 NVIDIA 显卡驱动，可以直接使用如下命令。

```
sudo apt-get install nvidia-384
```

A.3 安装 PyTorch

访问 PyTorch 官方网站 http://pytorch.org 就可以知道，PyTorch 对于安装的介绍非常人性化。

在“OS”“Package Manger”“Python”“CUDA”版本中选择一个类型，在下面的“Run this command”显示框中都会显示完整的命令信息，如图 A-10 所示。

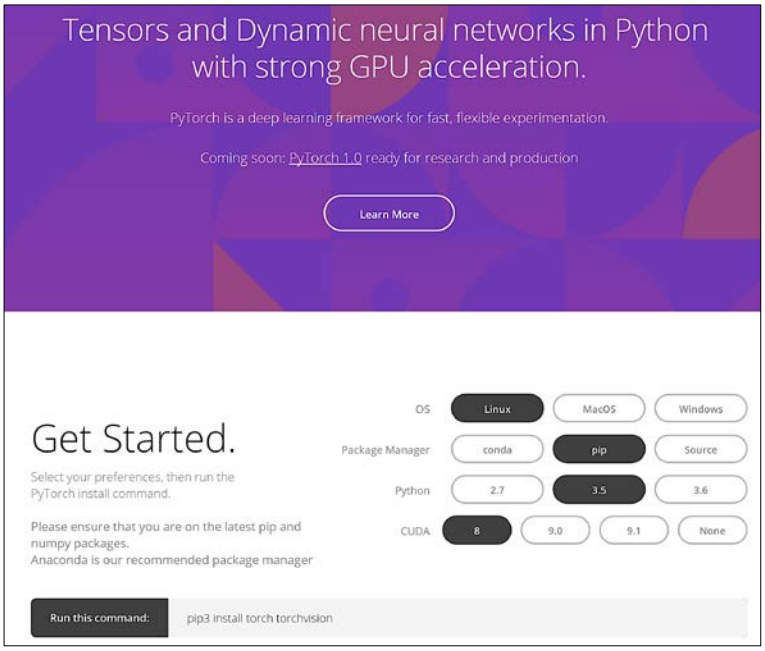


图 A-10 选择 PyTorch 安装版本（1）

基于使用方便的原因及个人习惯，如图 A-11 所示，我在这里安装的是 Linux 和 macOS 环境下的、pip 方式的、Python 3.5 版本的、CPU 版本的 PyTorch。

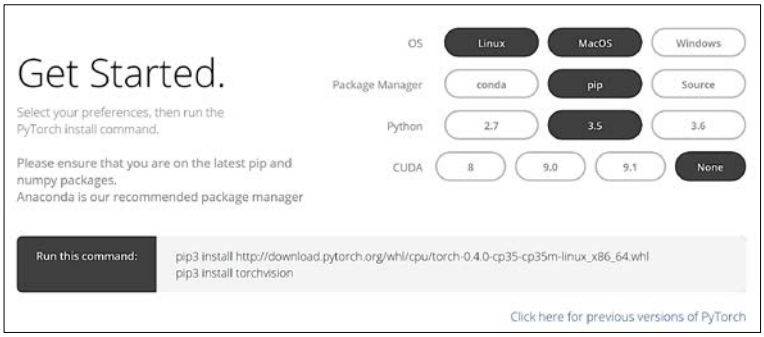


图 A-11　选择 PyTorch 安装版本（2）

相关命令如下。

```
pip3 install http://download.pytorch.org/whl/cpu/torch-0.3.1-cp36-cp36m-linux_x86_64.whl
pip3 install torchvision
```

安装过程执行完成，PyTorch 0.3.1 就安装好了。

如果希望安装 Linux 环境下的、pip 方式的、Python 3.5 版本的、CUDA 8.0 版本的 PyTorch，可以使用下面的命令。

```
pip3 install torch torchvision
```

这种方式是默认方式。

安装后，可以使用 Python Shell 进行验证，并导入 Torch 软件包。如果看到如图 A-12 所示的提示，就说明 PyTorch 已经正常安装了。

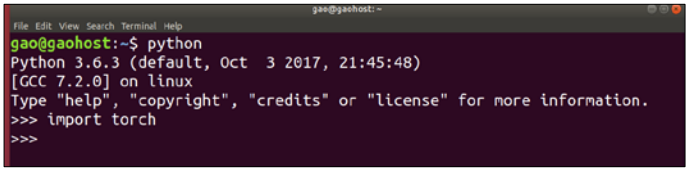

gao@gaohost: ~
File Edit View Search Terminal Help
gao@gaohost:~$ python
Python 3.6.3 (default, Oct 3 2017, 21:45:48)
[GCC 7.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>>

图 A-12　确认 PyTorch 安装状况

A.4　下载本书示例代码

如果需要下载本书的代码，请前往 GitHub 进行下载，地址为：

https://github.com/GAOYANGAU/DRLPytorch

在 Ubuntu 环境中，可以使用 git clone 命令来下载，具体如下。

```
cd ~

git clone https://github.com/GAOYANGAU/DRLPytorch
```

这样，代码就下载到当前用户的“Home”位置（~）了。

A.5 安装 PyCharm

PyCharm 是一个支持 Ubuntu 和 Windows 的 Python IDE 工具，多年以来一直受到 Python 程序员的好评。

安装此工具的方法非常简单。

A.5.1 方法一

前往 PyCharm 官方网站（https://www.jetbrains.com/pycharm/），单击“DOWNLOAD NOW”按钮，如图 A-13 所示。

图 A-13 PyCharm 官方网站

如图 A-14 所示，单击“Community”版本下的“DOWNLOAD”按钮。

保存文件，命令如下。

```
pycharm-community-2018.1.4.tar.gz
```

该文件将位于 Ubuntu 系统的如下位置，中间的用户名因人而异。

```
/home/gao/Downloads
```

将其移动到 /home/gao 目录下，在 Terminal 中输入下列命令。

```
mv /home/gao/Downloads/pycharm-community-2018.1.4.tar.gz /home/gao
```

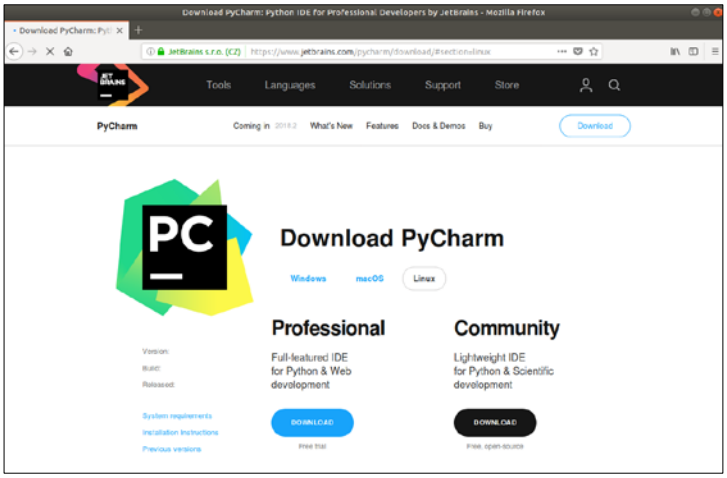


图 A-14　PyCharm 下载页面

解压该文件，命令如下。

```
cd /home/gao
tar zxvf pycharm-community-2018.1.4.tar.gz
```

给文件夹改名，命令如下。

```
mv pycharm-community-2018.1.4 pycharm
```

启动 PyCharm，命令如下。

```
cd /home/gao/pycharm/bin
./pycharm.sh
```

A.5.2　方法二

在 Ubuntu 16.04 以上的版本中，都可以用 Shell 命令直接进行安装，具体如下。

```
sudo snap install pycharm-community –classic
```

A.6　安装 Jupyter Notebook

Jupyter Notebook 是一个非常好的集编辑、运行、调试于一身的 Python IDE 环境，其在 GitHub 上的源代码中也包含 Jupyter Notebook 的代码版本供大家分段阅读与调试，非常方便。

Jupyter Notebook 的安装非常简单，只需要一行命令：

```
pip3 install jupyter
```

安装后，直接使用命令

```
jupyter notebook
```

即可启动 Jupyter Notebook，如图 A-15 所示。

图 A-15　*启动* Jupyter Notebook

如果出现报错信息

```
Error executing Jupyter command 'notebook': [Errno 2] No such file or directory
```

有两种方法可以解决。

一是修改 PATH 变量，具体如下。

```
Export PATH=$PATH:~/.local/bin/
```

二是使用 python 命令启动，具体如下。

```
python –m notebook
```

A.7　安装相关 Python 依赖包

A.7.1　安装 Box2D

安装 Box2D，命令如下。

```
pip3 install Box2D Box2D-kengz
```

A.7.2　安装 MuJoCo

如果要用本书中提到的深度强化学习在 Python 和 PyTorch 上的实验方法做相关实验的话，需要安装两个环境。一个是 MuJoCo，它是一个模拟环境，是用 C 和 C++ 编写的；另一个是 mujoco-py，它是 OpenAI 为 MuJoCo 开发的一个 Python 接口。

mujoco-py 安装非常方便，但 MuJoCo 的安装比较麻烦。在这里只演示申请 30 天试用版本的过程。

首先，如图 A-16 所示，访问 MuJoCo 官方网站（https://www.roboti.us/index.html），下载 MuJoCo 引擎。

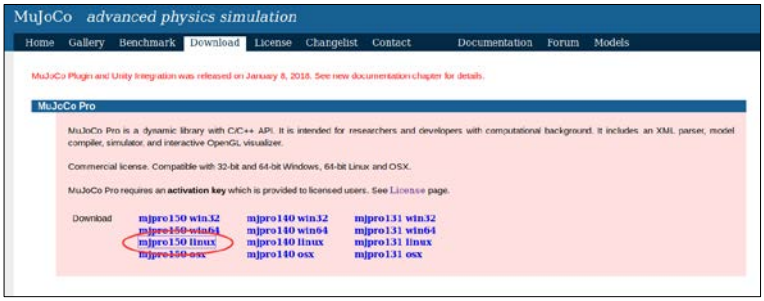

MuJoCo advanced physics simulation
Home Gallery Benchmark Download License Changelist Contact Documentation Forum Models
MuJoCo Plugin and Unity Integration was released on January 8, 2018. See new documentation chapter for details.
MuJoCo Pro
MuJoCo Pro is a dynamic library with C/C++ API. It is intended for researchers and developers with computational background. It includes an XML parser, model compiler, simulator, and interactive OpenGL visualizer.
Commercial license. Compatible with 32-bit and 64-bit Windows, 64-bit Linux and OSX.
MuJoCo Pro requires an activation key which is provided to licensed users. See License page.
Download
mjpro150 win32
mjpro150 win64
mjpro150 linux
mjpro150 osx
mjpro140 win32
mjpro140 win64
mjpro140 linux
mjpro140 osx
mjpro131 win32
mjpro131 win64
mjpro131 linux
mjpro131 osx

图 A-16　MuJoCo 官方网站

下载“mjpro150 linux”版本，就是 MuJoCo 1.5.0 版本。因为我们使用的是 Ubuntu，所以要下载 Linux 版本。

在 /home/gao/.mujoco/ 目录下解压 mjrp150_linux.zip 文件，具体如下。

```
unzip mjrp150_linux.zip
```

接下来，申请 MuJoCo Pro Trail License。

前往 Roboti 的官方网站（https://www.roboti.us/license.html#trial），如图 A-17 所示。

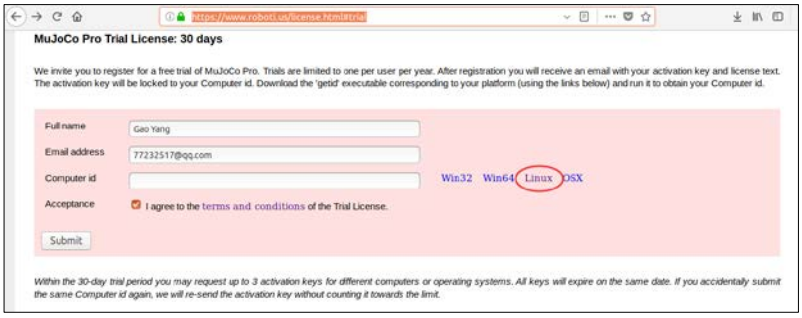

MuJoCo Pro Trial License: 30 days
We invite you to regsster for a free trial of MuJoCo Pro. Trials are limited to one per user per year. After registration you will receive an email with your activation key and license text. The activation key will be locked to your Computer id. Download the 'getid' executable corresponding to your platform (using the links below) and run it to obtain your Computer id.
Full name
Gao Yang
Email address
77232517@qq.com
Computer id
Win32 Win64 Linux OSX
Acceptance
I agree to the terms and conditions of the Trial License.
Submit
Within the 30-day trial period you may request up to 3 activation keys for different computers or operating systems. All keys will expire on the same date. If you accidentally submit the same Computer id again, we will re-send the activation key without counting it towards the limit.

图 A-17　申请 MuJoCo 许可的官方网站

单击右侧的“Linux”选项，下载一个 Linux 版本的 getid_linux（这是一个文件）。

修改这个文件的使用权限并执行它，具体如下，得到一个 Computer id。

```
chmod 755 gedit_linux
./gedit_linux
```

这时，我的计算机屏幕上出现了如图 A-18 所示的提示。

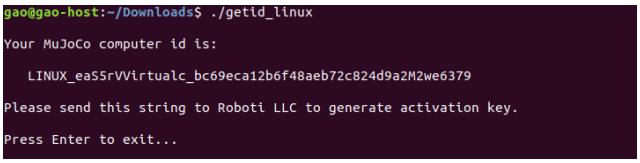

gao@gao-host:~/Downloads$ ./getid_linux
Your MuJoCo computer id is:
LINUX_eaS5rVVirtualc_bc69eca12b6f48aeb72c824d9a2M2we6379
Please send this string to Roboti LLC to generate activation key.
Press Enter to exit...

图 A-18　执行生成 Computer id

填入 Computer id，然后单击“Submit”按钮，如图 A-19 所示。

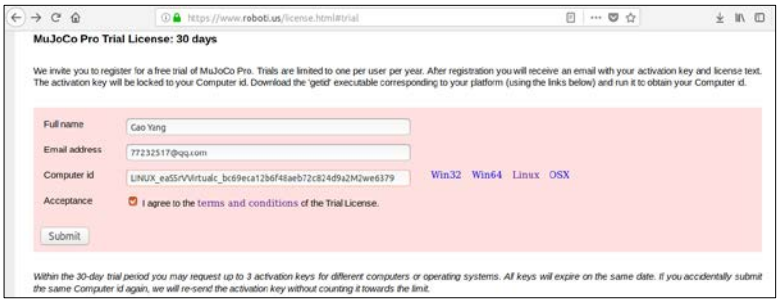

MuJoCo Pro Trial License: 30 days
We invite you to register for a free trial of MuJoCo Pro. Trials are limited to one per user per year. After registration you will receive an email with your activation key and license text. The activation key will be locked to your Computer id. Download the 'getid' executable corresponding to your platform (using the links below) and run it to obtain your Computer id.
Full name
Gao Yang
Email address
77232517@qq.com
Computer id
LINUX_eaS5rVVirtualc_bc69eca12b6f48aeb72c824d9a2M2we6379
Win32 Win64 Linux OSX
Acceptance
I agree to the terms and conditions of the Trial License.
Submit
Within the 30-day trial period you may request up to 3 activation keys for different computers or operating systems. All keys will expire on the same date. If you accidentally submit the same Computer id again, we will re-send the activation key without counting it towards the limit.

图 A-19　填入 Computer id

这时邮箱里就会收到两个文件，如图 A-20 所示。

图 A-20　收到的许可文件

下载 mjkey.txt，并把它复制到刚才解压文件时使用的文件夹中，如图 A-21 所示。

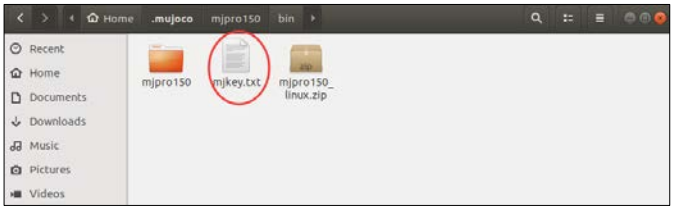


图 A-21　复制 mjkey.txt 文件

虽然邮件中有“copy the attached file 'mjkey.txt' to the 'bin' subdirectory”这样的提示，但还是要将文件放在 /home/gao/.mujoco 目录下。

接下来，安装 mujoco-py，命令如下。

```
pip3 install mujoco-py
```

安装 patchelf，要确认安装了当前 MuJoCo 所对应的 patchelf 0.9 版本，具体如下。

```
sudo apt install patchelf
```

安装 glfw3，具体如下。

```
sudo apt-get install libglfw3-dev
```

导出环境变量，具体如下。

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/.mujoco/mjpro150/bin
```

为了把这个变量设置得彻底一些，可以把它放到 .bashrc 中去，这样每次启动计算机以后就可以自动设置这个环境变量了，具体如下。

```
vim /home/gao/.bashrc
```

在尾部添加如下内容。

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/.mujoco/mjpro150/bin
```

安装相应的 XML 文件。

如果使用 pip 方式安装 mujoco-py，会有一个问题，就是一些关于模型的 XML 文件不存在。在做实验时，如果需要使用其中的 XML 模型文件，就会出现无法找到的问题。

建议进行如下操作，下载 mujoco-py 代码。

```
git clone https://github.com/openai/mujoco-py
```

下载后，找到里面的 xmls 文件夹，把它整体复制到 Python 3 的库文件夹下，具体如下。

```
cp –r /home/gao/mujoco-py/xmls /home/gao/.local/lib/python3.6/site-packages/
```

最后，确认在 Python 环境中能正常导入 mujoco_py 包，如图 A-22 所示。

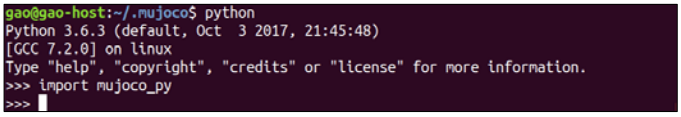

gao@gao-host:~/.mujoco$ python
Python 3.6.3 (default, Oct 3 2017, 21:45:48)
[GCC 7.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import mujoco_py
>>>

图 A-22　正常导入 mujoco-py 包

A.7.3　安装 SerpentAI

首先，安装 pyenv，命令如下。

```
curl -L https://github.com/pyenv/pyenv-installer/raw/master/bin/pyenv-installer |
bash
```

然后，添加环境变量，命令如下。

```
export PATH="~/.pyenv/bin:$PATH"
eval "$(pyenv init -)"
eval "$(pyenv virtualenv-init -)"
```

把以上三个 Shell 添加到 ~/.bashrc 尾部。

进行更新，命令如下。

```
pyenv update
```

接下来的步骤都是在 pyenv 环境中进行的。

安装 Python 3.6.5 和 OpenSSL Lib，命令如下。

```
sudo apt-get install libbz2-dev libssl-dev libreadline6 libreadline6-dev libsqlite3-
dev
```

```
pyenv install 3.6.5
```

官方文档建议安装 Python 3.6.4 或者更高版本，在这里安装的是 Python 3.6.5 版本。

创建一个 Python 3.6.5 的 pyenv 虚拟环境，命令如下。

```
pyenv virtualenv 3.6.5 serpent
```

创建一个供 SerpentAI 工程使用的文件夹，命令如下。

```
mkdir SerpentAI && cd SerpentAI
```

将 pyenv 虚拟环境指向这个文件夹，命令如下。

```
pyenv local serpent
```

SerpentAI 的 pyenv 环境设置成功后，提示符前会出现“(serpent)”字样，如图 A-23 所示。

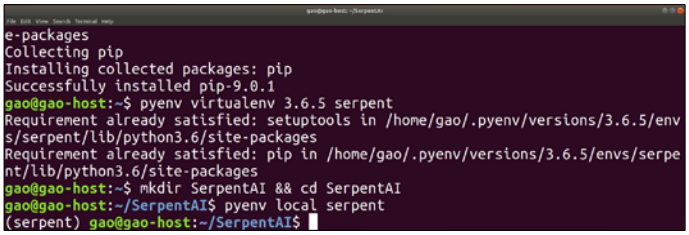


图 A-23　SerpentAI 的 pyenv 环境设置成功

在 pyenv 环境中继续安装 SerpentAI，命令如下。

```
pip3 install SerpentAI
```

安装其余依赖文件包，命令如下。

```
serpent setup
```

如图 A-24 所示，在安装过程中，屏幕上会出现一条小蛇的图案——“Serpent”的本意就是“蛇”，项目的名称和它的 Logo 是一致的，都使用了一条蛇的图案，并沿袭了 Linux 操作系统中用文字来码放 Logo 的习惯。

图 A-24　安装 SerpentAI

到这里，基本的 SerpentAI 组件都已经安装完毕了，我们可以用 SerpentAI 来做任何程序的接口代理了。不过，SerpentAI 官方建议安装下面几个组件，因此我们也要安装一下。

（1）OCR 组件

OCR（Optical Character Recognition，光学字符识别）是一个重要组件。如果你希望 SerpentAI 能帮你从输入的一帧图像中识别某个固定区域有没有字符或者数字，就要使用 OCR 组件。

OCR 组件的安装步骤很简单。先安装相应的依赖包，命令如下。

```
sudo apt-get install libtesseract-dev libleptonica-dev
```

直接使用命令，具体如下。

```
serpent setup ocr
```

（2）GUI 组件

GUI 组件是用来做可视化调试用的，它基于一个叫做 Kivy 的组件。

安装 GUI 组件，命令如下。

```
serpent setup gui
```

同时，安装 Kivy 的依赖包，命令如下。

```
sudo apt-get install -y \
    python-pip \
    build-essential \
    mercurial \
    git \
    python \
    python-dev \
    ffmpeg \
    libsdl-image1.2-dev \
    libsdl-mixer1.2-dev \
    libsdl-ttf2.0-dev \
    libsmpeg-dev \
    libsdl1.2-dev \
    libportmidi-dev \
    libswscale-dev \
    libavformat-dev \
    libavcodec-dev \
    zlib1g-dev
```

安装 Pygame，命令如下。

```
pip3 install pygame
```

安装 Redis，命令如下。

```
sudo apt-get install redis-server
```

在使用的时候，执行如下命令。

```
serpent visual_debugger
```

（3）ML 组件

ML 组件虽然不是 SerpentAI 的核心组件，但它是 SerpentAI 中一个非常关键的组件。在安装时，只需要输入如下命令。

```
serpent setup ml
```

系统会询问我们要安装的是 CPU 版本还是 GPU 版本，根据你的需要来输入就好了。

如果安装 CPU 版本，那么直接输入如下命令。

```
CPU
```

如果安装 GPU 版本，那么直接输入如下命令。

```
GPU
```

在此之后，SerpentAI 会启动安装 TensorFlow 1.4.1 版本（也可能是更新版本的 TensorFlow）。此外，如果使用的是 GPU 版本，要想正常使用 TensorFlow，就必须安装 NVIDIA 驱动、CUDA、cuDNN，三者缺一不可。

A.7.4 安装 Spritex

使用 git clone 命令下载 Spritex 在 GitHub 中的代码，具体如下。

```
git clone https://github.com/codetorex/spritex
```

安装 Cython，命令如下。

```
pip3 install Cython==0.23
```

安装 Kivy，命令如下。

```
pip3 install kivi==1.9.1
```

安装 Spritex，命令如下。

```
cd spritex

sudo ./install.sh
```

安装完成后，进行测试，命令如下，如图 A-25 所示。

```
spritex
```

图 A-25　成功安装 Spritex

这种方式可以让你在 Shell 模式下使用 Spritex。如果想在 Python 3 中使用 Spritex 的 API，建议使用 pip 进行安装，命令如下。

```
pip3 install spritex
```

A.7.5　安装 StarCraft Ⅱ

首先，安装 PySC2，命令如下。

```
pip3 install pysc2
```

然后，下载 StarCraft Ⅱ 在 Linux 中的引擎，这里使用的是 4.1.2 版本，命令如下。

```
curl -O http://blzdistsc2-a.akamaihd.net/Linux/SC2.4.1.2.60604_2018_05_16.zip
```

下载入口为 https://github.com/Blizzard/s2client-proto#downloads，如图 A-26 所示。

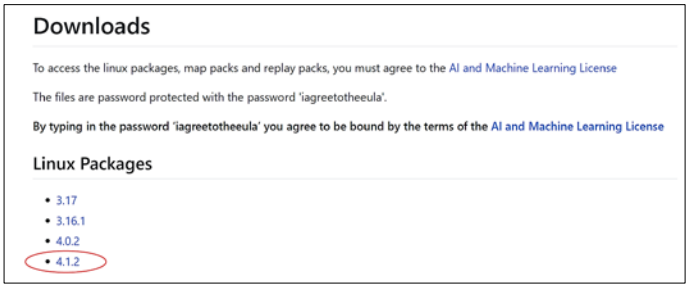

Downloads
To access the linux packages, map packs and replay packs, you must agree to the AI and Machine Learning License
The files are password protected with the password 'iagreetotheeula'.
By typing in the password 'iagreetotheeula' you agree to be bound by the terms of the AI and Machine Learning License
Linux Packages
3.17
3.16.1
4.0.2
4.1.2

图 A-26　StarCraft Ⅱ 的下载位置

如果不想使用 `curl -O` 命令，使用 Windows 的其他加速工具来下载也是可以的。这个文件有 4020MB，需要一段时间才能下载完。

解压 StarCraft Ⅱ 的文件，命令如下。

```
unzip SC2.4.1.2.60604_2018_05_16.zip
```

系统会询问解压密码，此时需要输入如下内容。

```
iagreetotheeula
```

然后，解压的所有文件都会保存在如下位置（请对应参考你的计算机屏幕上显示的内容）。

```
/home/gao/StarCraftII
```

下载地图文件 Melee，命令如下。

```
curl -O http://blzdistsc2-a.akamaihd.net/MapPacks/Melee.zip
```

解压 Melee 文件，命令如下。

```
unzip Melee.zip

mv Melee /home/gao/StarCraftII/Maps
```

测试 Agent 与环境，命令如下。

```
python -m pysc2.bin.agent --map Simple64
```

如果看到如图 A-27 所示的界面，说明 StarCraft Ⅱ 的环境已经安装好了，而且 PySC2 的环境也安装成功了。

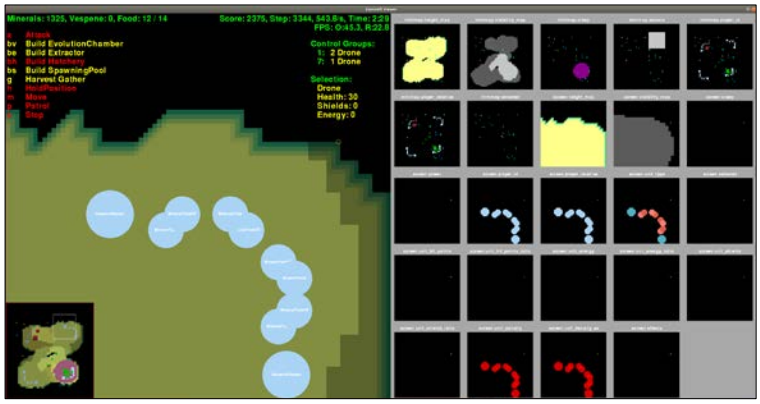


图 A-27　*启动一个随机的* Agent

此外，也可以让两个 Agent“互殴”，命令如下。

```
python -m pysc2.bin.agent --map Simple64 --agent2 pysc2.agents.random_agent.RandomAg
ent
```

还可以以人类玩家的身份启动游戏，命令如下。

```
python -m pysc2.bin.play --map Simple64
```

如图 A-28 所示，就是以人类玩家的身份玩 StarCraft Ⅱ。我截取了整个屏幕的一部分，和前面的几个画面一样，这就是 DeepMind 团队为爱好者们提供的一个 StarCraft 的简化版环境，并提供了丰富的接口。如果你从来没有玩过 StarCraft，很可能不知道现在屏幕上这些都是什么。标有“MineralField75”的是水晶矿，“Vespene Geyser”代表瓦斯矿（气矿），这两种物质就是游戏里的重要资源了。“SCV”是“Space Construction Vehicle”的缩写，直译过来就是“空间建筑车辆”，游戏玩家通常称其为“农民”，是用来采矿和建造建筑物的。

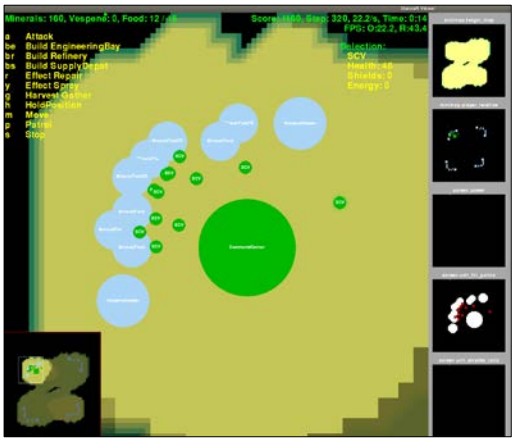


图 A-28　以人类玩家的身份玩 StarCraft Ⅱ

我试着在这个页面上玩了一下，互动倒是没什么问题，如图 A-28 所示那个被套圈的 SCV 就是我选中的“农民”，还能采矿呢……只不过，1 分钟都要不了，我就对游戏失去了兴趣……

DeepMind 帮我们做这样一个互动页面，好处还是大大的，那就是，不仅可以减少资源消耗，尤其是宝贵的显卡资源消耗，而且可以提高训练的节奏，这一点在 Agent 玩游戏的时候才会体现出来。

接下来，下载地图文件，如图 A-29 所示。

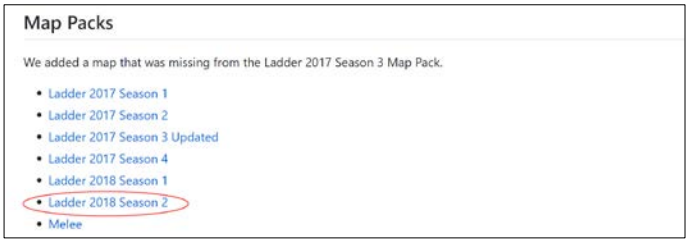

Map Packs
We added a map that was missing from the Ladder 2017 Season 3 Map Pack.
Ladder 2017 Season 1
Ladder 2017 Season 2
Ladder 2017 Season 3 Updated
Ladder 2017 Season 4
Ladder 2018 Season 1
Ladder 2018 Season 2
Melee

图 A-29 StarCraft Ⅱ 的地图文件

我选择下载 2018 Season2 的地图，命令如下。

```
curl -O http://blzdistsc2-a.akamaihd.net/MapPacks/Ladder2018Season2_Updated.zip
```

这个文件的下载入口同样在 https://github.com/Blizzard/s2client-proto#downloads。

最后，下载 Mini Game，命令如下。

```
curl -O https://github.com/deepmind/pysc2/releases/download/v1.2/mini_games.zip
```

A.7.6 安装 VizDoom

安装 ZDoom 依赖包，命令如下。

```
sudo apt-get install build-essential zlib1g-dev libsdl2-dev libjpeg-dev \
nasm tar libbz2-dev libgtk2.0-dev cmake git libfluidsynth-dev libgme-dev \
libopenal-dev timidity libwildmidi-dev unzip
```

安装 Boost Libraries，命令如下。

```
sudo apt-get install libboost-all-dev
```

安装 numpy，命令如下。

```
pip3 install numpy
```

克隆 VizDoom 的 GitHub 工程，命令如下。

```
git clone https://github.com/Marqt/ViZDoom
```

编译安装 VizDoom，命令如下。

```
cd ViZDoom
pip3 install .
```

最后，确认安装是否正确，启动 VizDoom 的 Basic Example。我是在这个位置克隆安装的，具体如下。

```
/home/gao/ml/ViZDoom
```

使用如下命令启动 ViZDoom。

```
cd /home/gao/ml/ViZDoom/examples/python
python basic.py
```

如果出现 ViZDoom 玩游戏的演示界面，就说明 ViZDoom 能够正常工作了，如图 A-30 所示。

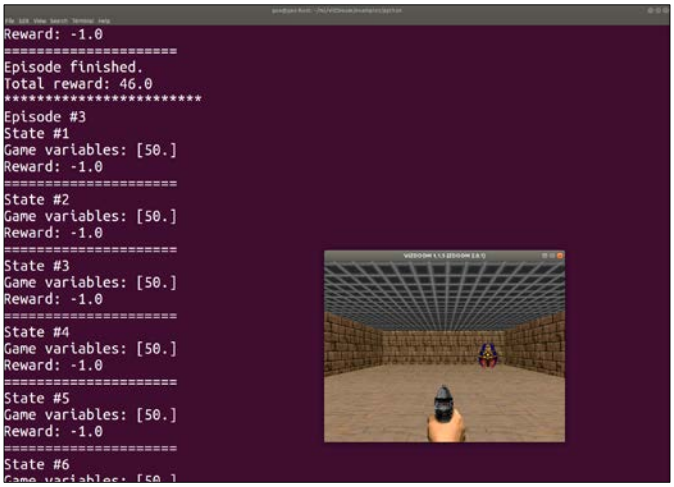


图 A-30　正常运行的 ViZDoom Basic 演示程序

A.8　安装 OpenCV

可以直接使用 apt 命令安装 OpenCV，具体如下，文件大概 300MB。

```
sudo apt-get install libopencv-dev python-opencv
```

安装 OpenCV 的 Python 库，命令如下。

```
pip3 install opencv-python
```

A.9　Python 语言简介

本节将简要介绍 Python 语言，对不熟悉 Python 语言的读者朋友会有一些帮助。

Python 是一种面向对象的解释型计算机程序设计语言，是由 Guido van Rossum 在 1989 年发明的，其第一个公开发行版发布于 1991 年。

Python 是纯粹的自由软件，它的源代码和解释器 CPython 遵循 GPL（GNU General Public License）协议。

Python 语言和 C 语言不一样，Python 是一种脚本语言。C 语言在写完源代码后，需要编译成

二进制代码才能够执行；Python 则不用，它在生产环境中仍旧是以源代码的 .py 文件形式出现的，在执行的瞬间才会由 Python 解释器将源代码转换为字节码，然后由 Python 解释器来执行这些字节码。

这种形式的好处是不用考虑平台系统的问题，可以和 Java 语言一样“一次编写，到处执行”。但其缺点也是显而易见的，就是每次进行字节码转换和字节码执行时，没有直接执行二进制代码的效率高。好在对于执行效率要求苛刻的场合毕竟比较少，另外，随着计算机硬件运算能力的提升，执行效率的矛盾也变得不明显了。

和其他计算机语言一样，Python 语言也有自己的一套语法，包括顺序、分支、循环、调用的程序组织结构，以及数字、字符串、列表、元组、集合等多种数据类型。我们在这里介绍一些在本书的实验中会使用的知识。

Python 语言有一个优点，就是随意性比较强。你几乎可以在程序的任何地方定义一个函数，也可以把一个 .py 文件写成一个类声明的形式。Python 程序可以是一个独立脚本的形式，也可以是“混搭”的，一个函数方法想返回什么就返回什么、想返回几个值就返回几个值……Python 语言几乎灵活到了“任性”的程度。我们几乎可以在任何安装了 Python 环境的系统中进行轻型和重型的调用。

不过，在工程界也有一句话，叫作“优点即缺点”。Python 这种“任性”的语言也会给我们带来很多问题，那就是：如果你不依靠相关模块的手册，或者不去查看手册代码，或者直接把相关内容打印出来，你简直没办法知道你将要访问的这个变量长什么样。

A.9.1 安装 Python

安装 Python 的方法不止一种，在 Ubuntu 17.10 上默认安装的是其 3.6.3 版本，可以执行如下命令查看。

```
gao@gao-host:~$ python -V
Python 3.6.3
```

A.9.2 Hello World

Python 的“Hello World”与其他计算机语言没什么区别，甚至更加简洁，可以直接在交互式编程环境中编写，具体如下。

```
print("Hello, Python!")
```

A.9.3 行与缩进

Python 脚本文件和普通的文本文件没有太大区别，一般我们喜欢将其写成 .py 文件的形式，示例如下。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
# 文件名：test.py

if True:
   print("True")
else:
print("False")
```

其中，“#”为注释标记。如果在一行中使用“#”，那么“#”后的内容是不会被解释执行的。下面的 if 和 else 是分支型语句，如果 if 后面的内容为 True（真实），则执行 if 所辖的部分，否则执行 else 所辖的部分。

在 Python 语言中，不使用 begin/end 或{}来表示执行段落的起止，这里的 if 和 else 需要靠左侧对齐，用缩进来表示段落所辖范围的界限。

A.9.4 变量类型

Python 语言中标准的数据类型有 Number（数字）、String（字符串）、List（列表）、Tuple（元组）、Dictionary（字典）。

如下代码演示了整数型数字、浮点型数字及字符串类型的赋值和打印操作。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

counter = 100 # 赋值整型变量
miles = 1000.0 # 浮点型
name = "John" # 字符串

print(counter)
print(miles)
print(name)
```

如下代码演示的是列表类型的操作。Python 语言中的列表很像 Java 语言中的数组，只不过列表允许不同类型的数据放在同一个列表中，而数组不可以（它要求所有元素类型一致）。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']

print(list) # 输出完整的列表
print(list[0]) # 输出列表中的第一个元素
print(list[1:3]) # 输出列表中的第二个至第三个元素
print(list[2:]) # 输出列表中从第三个元素开始至末尾的所有元素
print(tinylist * 2) # 输出列表两次
```

```
print(list + tinylist) # 打印组合的列表
```

如下代码演示的是元组类型的操作。操作方法和列表很像，但 Python 语法不允许对元组中的元素进行二次赋值。它相当于一个只读类型的列表。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
tinytuple = (123, 'john')

print(tuple) # 输出完整的元组
print(tuple[0]) # 输出元组中的第一个元素
print(tuple[1:3]) # 输出元组中的第二个至第三个元素
print(tuple[2:]) # 输出元组中从第三个元素开始至末尾的所有元素
print(tinytuple * 2) # 输出元组两次
print(tuple + tinytuple) # 打印组合的元组
```

如下代码演示的是字典类型的操作。字典类型有些像 Java 中的 HashMap，通过主键 Key 来访问对应的 Value 值（而不是靠下标来访问）。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

dict = {}
dict['one'] = "This is one"
dict[2] = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}

print(dict['one']) # 输出键为 'one' 的值
print(dict[2]) # 输出键为 2 的值
print(tinydict) # 输出完整的字典
print(tinydict.keys()) # 输出所有键
print(tinydict.values()) # 输出所有值
```

A.9.5 循环语句

如下代码演示的是 while 循环。while 循环后面的条件表示，在满足条件的时候，执行 `while` 所辖的程序段。

```
#!/usr/bin/python

count = 0
while (count < 9):
   print 'The count is:', count
   count = count + 1
```

```
print("Good bye!")
```

这段代码表示在 count < 9 的情况下，执行其下面的两行语句，但不包括

```
print("Good bye!")
```

这一行。

如下程序略显烦琐，不过内容还是很简单的。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

for num in range(10,20):  # 迭代 10 到 20 之间的数字
   for i in range(2,num): # 根据因子迭代
      if num%i == 0:      # 确定第一个因子
         j=num/i          # 计算第二个因子
         print('%d 等于 %d * %d' % (num,i,j))
         break            # 跳出当前循环
   else:                  # 循环的 else 部分
      print(num, '是一个质数')
```

这是循环的另一种写法——for 循环。for 循环也是一种循环，后面表示一个循环范围。在这段代码中有一个二重循环，也就是两个循环发生了嵌套（在一个循环的执行中有另一个循环）。外层循环是让 num 在 10 和 20 之间循环，内层循环是让 i 在 2 和 num 之间循环。

A.9.6 函数

函数是一种最小单位的代码段封装。

如以下代码所示，关键字是 def，def 后面的 printme 是函数名，str 是参数名称。这个函数的内容就是直接打印传入的变量值。最后两行就是对函数的调用。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

# 定义函数
def printme( str ):
   # 打印任何传入的字符串
   print(str)
   return

# 调用函数
printme("我要调用用户自定义函数!")
printme("再次调用同一函数")
```

A.9.7 模块

模块是一种大单位的代码段集合。例如，有一个 support.py 文件，其中有多个函数定义，有一个叫作 print_func。在不对 support.py 文件进行引用时，是不能调用 print_func 函数的。

在如下代码中，import support 表示导入 support.py 模块，support.print_func("Zara")表示调用 support 中的 print 函数，并传入变量 "Zara" 作为参数。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

# 导入模块
import support

# 现在可以调用模块里包含的函数了
support.print_func("Zara")
```

如下代码是导入模块的另一种写法，与以上方法的区别是，它能够导入一个模块的一部分而非全部。如下代码表示从 fib 这个模块中只导入 Fibonacci 这个函数。

```
from fib import Fibonacci
```

A.9.8 小结

以上就是在本书的实验中可能会用到的 Python 语言的基本语法。

至于强大的 Python 所支持的其他内容，大家如果有兴趣，可以找一些专门介绍 Python 的资料来学习，本书对 Python 基本语法的介绍就到此为止了。

A.10 本书涉及的主要开源软件版本

在本书的实验中涉及的一些软件和库信息，我在这里也进行了罗列（以下列出的都是软件包的名称）。在这些版本中测试示例代码，都是可以通过的。

（1）操作系统

- ubuntu 17.10

（2）Python

- python 3.6.3

（3）经由 pip3 安装的软件

- numpy 1.4.2
- notebook 5.5.0

- jupyter 1.0.0
- jupyter-core 4.4.0
- gym 0.10.5
- graphviz 0.8.3
- matplotlib 2.2.2
- Box2D 2.3.2
- Box2D-kengz 2.3.3
- Mujoco-py 1.5
- pygame-1.9.3
- Cython 0.23

（4）其他第三方开源软件（使用 apt 安装或一般下载安装）

- python3-tk 3.6.3
- python3-pip 9.0.1
- swig 3.0.2
- MuJoCo 1.5.0
- Patchelf 0.9
- libglfw3-dev 3.2.1
- pyenv 1.2.5
- redis-server 4:4.0.1-7
- kivy 1.10.1/1.9.1（for Spritex）

参考文献

[1] 郭宪, 方勇纯. 深入浅出强化学习：原理入门. 电子工业出版社.

[2] 高扬, 卫峥, 尹会生. 白话大数据与机器学习. 机械工业出版社.

[3] 高扬, 卫峥. 白话深度学习与 TensorFlow. 机械工业出版社.

[4] 高扬. 数据科学家养成手册. 电子工业出版社.

[5] Ian Goodfellow. 深度学习. 人民邮电出版社.

[6] 陈宝林. 最优化理论与算法（第 2 版）. 清华大学出版社.

[7] 廖星宇. 深度学习入门之 PyTorch. 电子工业出版社.

[8] 陈云. 深度学习框架 PyTorch：入门与实践. 电子工业出版社.

[9] 冯超. 强化学习精要：核心算法与 TensorFlow 实现. 电子工业出版社.

[10] 康崇禄. 蒙特卡罗方法理论和应用. 科学出版社.

[11] Qinglai Wei, Ruizhuo Song, Benkai Li, Xiaofeng Lin. Self-Learning Optimal Control of Nonlinear Systems. 科学出版社.

[12] 那不勒坦. 算法基础（第 5 版）. 贾洪峰, 译. 人民邮电出版社.

[13] 彭伟. 揭秘深度强化学习. 中国水利水电出版社.

[14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, Demis Hassabis. Human-level control through deep reinforcement learning. 2015.

[15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. 2013.

[16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. 2016.

[17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, Demis Hassabis. Mastering the game of Go without human knowledge. 2017.

[18] Chris J. Maddison, Aja Huang, Ilya Sutskever, David Silver. Move Evaluation In Go Using Deep Convolution Neural Networks. 2015

[19] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. 2016.

[20] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, Daan Wierstra. Continuos control with deep reinforcement learning. 2016.

[21] Hado van Hasselt, Arthur Guez, David Silver. Deep Reinforcement Learning with Double Q-Learning. 2015.

[22] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas. Dueling Network Archietectures for Deep Reinforcement Learning. 2016.

[23] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, IlyaSutskever. Evolution Strategies as a Scalable Alternative to Reinforcement Learning. 2017.

[24] Lianmin Zheng, Jiacheng Yang, Han Cai, Weinan Zhang, Jun Wang, Yong Yu. MAgent: A Many-Agent Reinforcement Learning Platform for Artificial Collective Intelligence. 2017.

[25] Tomas H. Maul, Andrzej Bargiela, Chong Siang Yew, Abdullahi S. Adamu. Towards Evolutionary Deep Nearual Networks. 2014

[26] Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z Leibo, David Silver, Koray Kavukcuoglu. Reinforcement Learning With Unsupervised Auxiliary Tasks. 2016.

[27] Kenneth O. Stanley, Risto Miikkulainen. Efficient Evolution of Neural Network Topologies. 2002.

[28] Marc Lanctot, Vinicius Zambaldi, Audrunas Gruslys, Angeliki Lazaridou, Karl Tuyls, Julien Pérolat, David Silver, Thore Graepel. A Unified Game-Theoretic Approach to Multiagent Reinforcement Learning. 2017.

[29] Samuel Alvernaz, Julian Togelius. Autoencoder-augmented Neuroevolution for Visual Doom Playing. 2017.

[30] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros. Curiosity-driven Exploration by Self-supervised Prediction. 2017.

[31] Alexey Dosovitskiy, Vladlen Koltun. Learning to Act by Predicting the Future. 2017.

[32] Yuxin Wu, Yuandong Tian. Training Agent for First-Person Shooter Game with Actor-Critic Curriculum Learning. 2017.

[33] Devendra Singh Chaplot, Guillaume Lample. Arnold: An Autonomous Agent to play FPS Games. 2016.

[34] Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, Wojciech Jaśkowski. ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning. 2016.

[35] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, W. Philip Kegelmeyer. SMOTE: Synthetic Minority Over-sampling Technique. 2002.

[36] Jack Shannon, Marek Grzes. Reinforcement Learning using Augmented Neural Networks. 2018.

[37] Hima Varsha Dureddy, Zachary Kaden. Handling Cold-Start Collaborative Filtering with Reinforcement Learning. 2018.

[38] Wenjia Meng, Qian Zheng, Long Yang, Pengfei Li, Gang Pan. Qualitative Measurements of Policy Discrepancy for Return-based Deep Q-Network. 2018.

[39] Junchi Liang, Abdeslam Boularias. Task-Relevant Object Discovery and Categorization for Playing First-person Shooter Games. 2018.

[40] Hado van Hasselt. Double Q-learning. 2010.

[41] Song Han, Jeff Pool, John Tran, William J. Dally. Learning both Weights and Connections for Efficient Neural Networks. 2015.

[42] Song Han, Huizi Mao, William J. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. 2015.

[43] Wei Pan, Hao Dong, Yike Guo. DropNeuron: Simplifying the Structure of Deep Neural Networks. 2016.

[44] Suraj Srinivas, R. Venkatesh Babu. Learning Neural Network Architectures using Backpropagation. 2016.

[45] Yunchao Gong, Liu Liu, Ming Yang, Lubomir Bourdev. Compressing Deep Convolutional Networks using Vector Quantization. 2014.

[46] Jonghoon Jin, Aysegul Dundar, Eugenio Culurciello. Flattened Convolutional Neural Networks for Feedforward Acceleration. 2015.

[47] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. 2015.

[48] Tim Salimans, Diederik P. Kingma. Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks. 2016.

[49] Alfredo V. Clemente, Humberto N. Castejon, Arjun Chandra. Efficient Parallel Methods For Deep Reinforcment Learning. 2017.

[50] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas, Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray, Kavukcuoglu, David Silver. Massively Parallel Methods for Deep Reinforcement Learning. 2015.

[51] Tom Schaul, John Quan, Ioannis Antonoglou, David Silver. Priorized Experience Replay. 2016.

[52] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, Martin Riedmiller.

Deterministic Policy Gradient Algorithms. 2014.

[53] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, Igor Mordatch. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments. 2018.

[54] Xiangxiang Chu, Hangjun Ye. Parameter Sharing Deep Deterministic Policy Gradient for Cooperative Multi-agent Reinforcement Learning. 2017.

[55] Vijay R. Konda, John N. Tsitsiklis. Actor-Critic Algorithms. 2000.

[56] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore,Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. 2017.

[57] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, Piotr Dollar. Microsoft COCO: Common Objects in Context. 2015.

[58] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, Pieter Abbeel. Trust Region Policy Optimization. 2017.

[59] Pieter Abbeel, Andrew Y. Ng. Apprenticeship Learning via Inverse Reinforcement Learning. 2004.

[60] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. 2017.